Treball de Fi de Grau

Enginyeria en Tecnologies Industrials

SLAM Visual basat en l'ús de Bundle Adjustment

MEMÒRIA

Autor: Oriol Vendrell Gallart
Director: Joan Solà Ortega
Ponent: Joan Rosell Gratacòs

Data: Juny 2019

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Resum

L'objectiu d'aquest treball de fi de grau és desenvolupar un sistema per fer localització i cartografia (conegut com a *Simultaneous Localization and Mapping* o SLAM en robòtica) basat en un algoritme de bundle adjustment, que com el seu nom indica consisteix en fer un ajust de feixos per tal d'afinar l'estructura de l'escena visual. Per fer-ho, es farà servir una càmera monocular per obtenir una seqüència d'imatges a processar.

En primer lloc s'ha fet una anàlisi i selecció d'una combinació de tríada detector-descriptor-matcher amb les opcions implementades a la llibreria de visió per computador OpenCV. S'ha fet una anàlisi d'una seqüència de vídeo per obtenir tracks entre imatges consecutives i s'ha obtingut l'opció més adient al propòsit del projecte.

Ens segon lloc, s'ha partit de la llibreria en C++ anomenada Wolf que s'està desenvolupant a l'IRI (Institut de Robòtica i Informàtica Industrial) per resoldre problemes de localització en robòtica i s'ha implementat un algoritme de bundle adjustment per fer localització i cartografia amb les imatges d'una càmera. Es fa un seguiment o track de landmarks (que corresponen a punts en l'espai tridimensional) en imatges consecutives a temps real i s'estimen els paràmetres d'estat resolent un problema de *graph* SLAM amb optimització no lineal.

Per últim, es fa un node de ROS (*Robot Operating System*) que implementa el sistema desenvolupat i es fa una avaluació del funcionament de l'algoritme a partir de les estimacions en la solució del problema i de la visualització.

Agraïments

En primer lloc, voldria manifestar el meu agraïment a en Joan Solà, qui ha estat el director d'aquest treball, per guiar-me en aquest viatge al llarg de quatre mesos, pel suport rebut i per l'oportunitat d'iniciar-me en el món de la robòtica mòbil.

En segon lloc, voldria agraïr el suport rebut d'en Joan Vallvé en el desenvolupament del projecte i l'ajuda d'en Joaquim Casals en els dubtes que m'han anat sorgint.

També voldria donar les gràcies als membres del *Mobile Robotics Group* de l'IRI (Institut de Robòtica i Informàtica Industrial) i en especial als membres del projecte Wolf per l'acollida, els moments i l'ajuda rebuda.

Per últim, un agraïment a la meva família que sempre són una inspiració per mi.

Sumari

Capítol 1

Introducció

L'habilitat de localitzar-se dins d'un entorn és una capacitat que els éssers humans podem desenvolupar amb destresa i, de fet, és un procés que fem de manera intuïtiva cada vegada que, per exemple, entrem en un lloc desconegut. Només a partir de la visió, som capaços de veure on estan situats elements característics en el mapa (*landmarks*) i tenir una noció d'on estem dins d'aquest espai; i encara ens és més fàcil si hi afegim elements del sistema propioceptiu com ho és el sentit de l'equilibri, que ens dona un complement perfecte als sentits exteroceptius. En robòtica, aquesta habilitat de localitzar-se en un entorn s'intenta reproduir amb l'ús de sensors que imiten els sentits humans, en un dels numerosos intents dels humans d'imitar la perfecció de la naturalesa. Aquesta capacitat dins la natura té un grau de complexitat diferent segons l'espècie, ja que hi ha animals amb sistemes vestibulars i visuals molt desenvolupats que superen als humans amb escreix, però en el camp artificial, podríem dir que encara superem la màquina, perquè per a un robot aquest procés no és gens trivial. En aquest projecte es treballarà en la direcció de dotar a un robot d'aquesta capacitat per tal que pugui desenvolupar aquesta tasca de la forma més acurada possible.

1.1 Objectius

L'objectiu d'aquest projecte és desenvolupar un algoritme de localització i cartografia per solucionar el problema conegut en el camp de la robòtica com SLAM (*simultaneous localization and mapping*): conèixer com és el mapa del món exterior i com es mou el robot situat dins d'aquest món. El propòsit és solucionar el problema aplicant bundle adjustment, que tal com diu el nom consisteix en ajustar feixos de raigs (feixos de llum que entren a la càmera). Es farà amb l'input de mesures visuals d'una càmera monocular en una seqüència d'imatges. Per aconseguir completar aquest 'puzzle' complex, el projecte s'ha dividit en diferents parts per abordar el problema des dels diferents elements que el composen.

Primerament, quan es pensa en mesures d'una càmera, el primer que ve a la ment és una imatge, però per la màquina una imatge no és més que un munt de números sense sentit. Per tant, la primera necessitat és saber com obtenir les mesures de la càmera i com donar-los sentit pel propòsit del projecte. En aquest sentit, el primer objectiu és seleccionar una tríada de *detector*, *descriptor* i *matcher* per detectar, descriure i fer match al llarg d'una seqüència d'imatges.

Seguidament, un cop es tenen les mesures visuals, es necessita processar-les per tal de donar resposta al problema formulat i reconstruir el mapa del món exterior i la trajectòria.

En aquest sentit, el segon objectiu del projecte és contribuir a una llibreria en C++ per resoldre problemes de localització i cartografia anomenada WOLF a partir del desenvolupament d'un algoritme que apliqui bundle adjustment per processar la informació de la càmera i definir l'estructura del món exterior (*landmarks*) i la localització dins d'aquest món.

1.2 Abast del projecte

Aquest projecte es desenvolupa dins el marc de la feina feta al grup de robòtica mòbil de l'Institut de Robòtica i Informàtica Industrial (IRI – CSIC, UPC). Pel que fa al desenvolupament de l'algoritme per solucionar el problema de localització, no es desenvoluparà des de zero, es partirà del treball fet a Wolf i de les eines ja existents per implementar-lo. La part matemàtica de solucionar el problema es delegarà a un *solver* extern (es farà ús de *Ceres*).

Referent a la tríada *detector-descriptor-matcher*, no es pretén crear un mètode nou per fer aquestes tasques. Es partirà d'*OpenCV*, que és una llibreria per fer visió per computador, i es treballarà en triar una combinació adequada pels propòsits del projecte dels algoritmes que proporciona aquesta llibreria.

Per últim, es farà experimentació dels resultats fent ús d'una càmera, però no entra en l'abast del projecte integrar la feina en un robot amb múltiples sensors de diferents tipus.

Capítol 2

Bundle Adjustment

En aquest capítol es descriurà el problema de bundle adjustment i es farà un enfocament als elements que fan falta per resoldre el problema.

2.1 Bundle Adjustment

El bundle adjustment és el problema que consisteix en ajustar l'estructura dels elements del mapa 3D (núvol de punts) al mateix temps que els paràmetres de visió, ja sigui la calibració de la càmera o la *pose* de la càmera en el mapa [1]. En el cas d'aquest projecte, com que l'objectiu és fer SLAM visual, s'estimaran al mateix temps les localitzacions del núvol de punts 3D i la trajectòria de la càmera (*poses* durant el moviment).

Tant en la reconstrucció del núvol de punts com de la *pose* de la càmera, es troba una estimació òptima minimitzant una funció de cost que quantifica l'error de l'estimació.

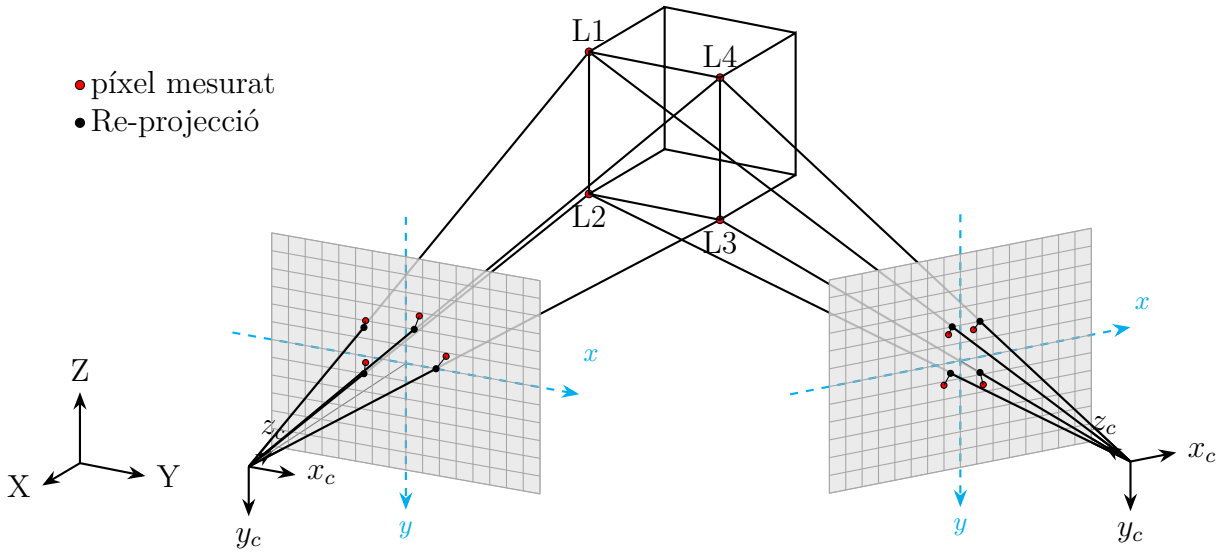

Figura 2.1: Representació de la problemàtica a solucionar aplicant Bundle Adjustment. El píxel de la imatge i la re-projecció del punt 3D sobre la imatge no coincideixen.

Tal i com es pot veure en la Figura 2.1, la problemàtica és que les mesures en les

imatges dels punts 3D o *landmarks* de l'exterior no coincideixen amb la mesura que s'obté quan amb el model de projecció de la càmera es re-projecta la posició del punt tridimensional sobre el pla de la imatge. Aquesta diferència és un error que s'anomena error de re-projecció i que es pot modelitzar:

$$\text{Píxel mesurat: } \mathbf{z} = \begin{bmatrix} m_x \\ m_y \end{bmatrix} \tag{2.1}$$

$$\text{Re-projecció punt 3D: } \begin{bmatrix} u \\ v \\ w \end{bmatrix} = K {}^{W}R_{C}^{T}(L - {}^{W}T_{C}) \tag{2.2}$$

on K és la matriu de paràmetres intrínsecs, R i T fan referència a la composició del sistema de referència de la càmera en el sistema global, i L és el punt 3D o landmark. L'error de re-projecció és doncs

$$\mathbf{e}(x) = \begin{bmatrix} m_x \\ m_y \end{bmatrix} - \begin{bmatrix} \frac{u}{w} \\ \frac{u}{w} \end{bmatrix} \tag{2.3}$$

L'error de re-projecció $e(x)$ depèn de l'estat x, que és un vector format per tots els paràmetres que s'han d'estimar, que són la posició del punt 3D i la posició i orientació de la càmera, és a dir:

$$\mathbf{e}(x) = [R, T, L] \tag{2.4}$$

Com a nota, cal comentar que en comptes de treballar amb la matriu de rotació R es pot treballar amb quaternions, aconseguint una reducció de 9 a 4 variables.

Per ajustar aquestes variables i minimitzar la suma de tots els errors de re-projecció per cada landmark i captura de la càmera en el seu moviment, cal minimitzar la funció d'error respecte les variables de l'estat $\mathbf{x}$ per tal de trobar-ne una estimació òptima.

Aquest problema és no lineal ja que en fer la re-projeció del punt tridimensional s'obté un punt bidimensional en coordenades homogènies, i per tal de fer la resta entre la mesura i la re-projecció cal dividir pel 3r component. Com que aquesta component depèn de l'estat $\mathbf{x}$, el problema no és lineal. A banda d'això, hi ha la rotació $\mathcal{R}$ que tampoc és lineal.

A partir d'aquí es poden aplicar diferents mètodes per optimitzar l'estat. En les següents seccions d'aquest capítol s'exposarà com es formula el problema d'SLAM i com es pot solucionar aplicant bundle adjustment, ja que solucionant aquest problema s'estima l'estat propi i el de l'entorn, cosa que és justament el que pretén resoldre un problema d'SLAM.

2.2 SLAM

El que en robòtica es coneix com a *simultaneous localization and mapping* o, de forma abreviada, SLAM, és el problema d'estimar l'estructura del món (el mapa) mentre al mateix temps s'estima la localització pròpia dins d'aquest mapa. En aquest apartat s'expliquen els fonaments de l'SLAM basats en la guia d'en Joan Solà d'introducció a l'SLAM [2].

En aquest problema els dos elements principals són el robot (un vehicle mòbil) que està equipat amb algun tipus de sensor i el mapa format per landmarks. El robot travessa un espai desconegut i pot realitzar mesures del seu propi moviment i mesures detectant objectes en el medi que l'envolta. Amb aquestes mesures construeix el mapa que l'envolta i al mateix temps utilitza el mapa i les mesures per localitzar-s'hi.

El problema es pot resumir en tres passos que es repeteixen de forma iterativa:

– El robot $\mathcal{R}$ es mou, i degut al soroll i als errors, creix la incertesa de la localització del robot. El model matemàtic d'aquest moviment que relaciona dos estats s'anomena *model de moviment*:

$$\mathcal{R} \leftarrow f(\mathcal{R}, \mathbf{u}, \mathbf{n}) \tag{2.5}$$

L'estat del robot $\mathcal{R}$ s'actualitza quan el robot es mou en funció del senyal de control $\mathbf{u}$ i de la pertorbació $\mathbf{n}$.

– El robot $\mathcal{R}$ descobreix *features* interessants a l'entorn, que s'anomenen *landmarks* $\mathcal{L}_i$, i que són incorporats al mapa $\mathcal{M}$. A causa d'error en els sensors $\mathcal{S}$ exteroceptius, la localització dels landmarks serà incerta. El model matemàtic que determina la posició dels landmarks a partir de les mesures $\mathbf{z}_i$ del sensor s'anomena *model d'observació invers*:

$$\mathcal{L}_i = g(\mathcal{R}, \mathcal{S}, \mathbf{z}_i) \tag{2.6}$$

L'estat del landmark $\mathcal{L}_i$ és funció de l'estat del robot $\mathcal{R}$, del sensor $\mathcal{S}$ i de la mesura $\mathbf{z}_i$.

– El robot $\mathcal{R}$ observa landmarks $\mathcal{L}_j$ que ja havien estat observats, i els fa servir per corregir al mateix temps la localització pròpia i la dels landmarks. En aquest cas la incertesa decreix. El model matemàtic per predir els valors de la mesura $\mathbf{z}_j$ a partir de la localització del landmark i del robot s'anomena *model d'observació directe*:

$$\mathbf{z}_j = h(\mathcal{R}, \mathcal{S}, \mathcal{L}_j) \tag{2.7}$$

La mesura esperada $\mathbf{z}_j$ és funció de l'estat del robot $\mathcal{R}$, del sensor $\mathcal{S}$ i de l'estat del landmark $\mathcal{L}_j$. Idealment la funció g() hauria de ser la inversa de h(), però en alguns casos com l'SLAM monocular la mesura obtinguda no té la informació de tots els graus de llibertat del landmark i és no invertible.

En el problema d'SLAM, les diferents entitats que hi estan involucrades són el robot $\mathcal{R}$, el mapa $\mathcal{M}$, els landmarks $\mathcal{L}_i$, el sensor $\mathcal{S}$ que té un soroll associat a la mesura i les mesures $\mathbf{z}_i$ del sensor. Cada parella sensor-landmark defineix una observació.

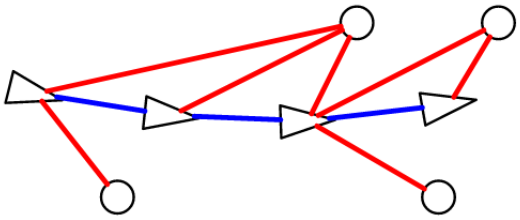

Figura 2.2: Representació del problema d'SLAM. Els triangles representen els estats del robot $\mathcal{R}$, les línies blaves mesures de moviment, els cercles representen landmarks $\mathcal{L}$ del mapa $\mathcal{M}$, i les línies vermelles les mesures dels landmarks. Font: [3]

Amb tots aquests models definits i una eina d'estimació (*solver*) es pot construir una solució al problema de localització. Per solucionar el problema hi ha diferents mètodes: per una banda, el Kalman filter (EKF), i per l'altra l'SLAM basat en factor graph, també anomenat graph slam. Del Kalman filter (EKF), per als interessats, se'n pot trobar material extens a [2] o [4]. Del factor graph se'n fa una breu introducció a la secció 2.3, ja que és el mètode que es fa servir en el projecte.

2.3 Factor Graph

El problema de l'SLAM, introduït a la secció 2.2, es pot entendre també com una xarxa de relacions o un *graph* [5] . En aquesta secció es fa una introducció a les bases de l'SLAM entès com un factor graph, que bàsicament és un resum dels continguts del curs d'SLAM d'en Joan Solà [3].

El procés de l'SLAM es pot representar com una *Dynamic Bayes Network* (DBN), que és un model probabilístic gràfic que representa un conjunt de variables aleatòries i les seves dependències condicionals a partir d'un graph dirigit acíclic. La dependència condicional es marca amb una fletxa dirigida connectant dues variables (dos nodes en el

graph), i per tant $A \leftarrow B$ indica que A està condicionat per B, és a dir, A depèn de B. La propietat de ser 'acíclic' garanteix que no s'entra en bucles de dependència.

En el problema de l'SLAM, es tenen quatre tipus diferents de variable aleatòria, que són vectors d'estats:

$$\begin{aligned} \text{Estats del robot: } X &= \{\mathbf{x}_i\}\ ,\ i \in 0 \cdots M \\ \text{Estats dels landmarks: } L &= \{\mathbf{l}_j\}\ ,\ j \in 1 \cdots N \\ \text{Controls del robot: } U &= \{\mathbf{u}_i\}\ ,\ i \in 1 \cdots M \\ \text{Mesures dels landmarks: } Z &= \{\mathbf{z}_k\}\ ,\ k \in 1 \cdots K \end{aligned}$$

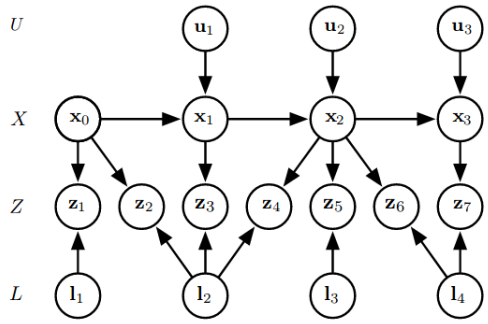

Figura 2.3: Dynamical Bayes Network per un sistema d'SLAM basat en landmarks. Representa el problema en la Figura 2.2. Font: [3]

En aquest problema els estats X i L no són directament observables, però es tracta d'estimar el seu valor a partir de comparar les mesures dels sensors amb l'esperança de la mesura donat els models matemàtics i les variàncies de cada mesura (per ponderar la precisió del sensor).

Tornant als models matemàtics que intervenen en l'SLAM i dels quals s'ha parlat en la secció 2.2, els models es poden formular amb la següent notació:

$$\text{Model de moviment: } \mathbf{x}_i = f_i(\mathbf{x}_{i-1}, \mathbf{u}_i) + \mathbf{n}_i, \quad \mathbf{n}_i \sim \mathcal{N}\{0, \Omega_i^{-1}\} \tag{2.8}$$

$$\text{Model d'observació directe } \mathbf{z}_k = h_k(\mathbf{x}_{i_k}, \mathbf{l}_{j_k}) + \mathbf{v}_k, \quad \mathbf{v}_k \sim \mathcal{N}\{0, \Omega_k^{-1}\} \tag{2.9}$$

considerant els sorolls associat al control $\mathbf{n}_i$ i a la mesura del sensor $\mathbf{v}_k$ com a variables Gaussianes amb unes covariàncies respectivament Ω_i^{-1} i Ω_k^{-1}.

En la Dynamic Bayes Network aquests models es poden posar com un sub-graph que representa la probabilitat condicional del valor futur de la variable després d'actualitzar-la en funció dels paràmetres de $f(\cdot)$ i $h(\cdot)$:

$$P(\mathbf{x}_i|\mathbf{x}_{i_{-1}}, \mathbf{u}_i) \qquad \text{i} \qquad P(\mathbf{z}_k|\mathbf{x}_i, \mathbf{l}_j) \tag{2.10}$$

Amb aquesta formulació es pot escriure la probabilitat conjunta de la trajectòria, el

mapa format per landmarks, les mesures i el control com a:

$$P(X, L, U, Z) \propto P(\mathbf{x}_o) \prod_{i=1}^{M} P(\mathbf{x}_i|\mathbf{x}_{i_{-1}}\mathbf{u}_i) \prod_{k=1}^{K} P(\mathbf{z}_k|\mathbf{x}_i, \mathbf{l}_j) \tag{2.11}$$

on $P(\mathbf{x}_o)$ representa el *prior* de la localització inicial del robot. L'objectiu del *solver* serà trobar les variables d'estat X^* i L^* que maximitzen aquesta probabilitat,

$$\{\mathbf{X}^*, \mathbf{L}^*\} = \arg\max_{X,L} P(\mathbf{x}_o) \prod_{i=1}^{M} P(\mathbf{x}_i|\mathbf{x}_{i_{-1}}\mathbf{u}_i) \prod_{k=1}^{K} P(\mathbf{z}_k|\mathbf{x}_i, \mathbf{l}_j) \tag{2.12}$$

En el problema de l'SLAM considerant que el mapa està format de landmarks, en el graph es poden distingir dos tipus de nodes (veure Figura 2.4):

- **Nodes de variables**: constitueixen els estats. Són els paràmetres desconeguts que es volen estimar.

- **Factor nodes**: constitueixen les restriccions entre els estats. Venen de les mesures dels sensors i són informació coneguda.

Amb aquesta classificació i a termes de calcular l'error associat a cada mesura, la distinció entre estats del robot i landmarks no és important, així com tampoc la distinció entre moviment del robot i mesures externes.

Per tant, els estats tant dels landmarks com del robot constitueixen un vector d'estats $\mathbf{x}$ amb N blocs.

$$\mathbf{x} = [\mathbf{x}_1 \dots \mathbf{x}_N]^T \tag{2.13}$$

I les observacions, ja siguin de landmarks o el propi moviment, també es poden agrupar en un bloc de $\mathbf{K}$ elements ($\mathbf{k} \in \mathbf{K}$), havent-hi un factor per cadascun d'ells.

$$\mathbf{z} = [\mathbf{z}_1 \dots \mathbf{z}_K]^T \tag{2.14}$$

Cada factor $\phi_\mathbf{k}$ captura la informació que entra al graph i que es propaga per la resta d'estats que es volen estimar. Per cada factor es tenen els índexs dels estats que relaciona, el model d'observació i la variància.

En el mètode de resoldre el problema a partir del factor graph, per a cada factor es pot calcular l'error associat a cada observació segons sigui un moviment o un landmark $\mathbf{e}_k$ de la forma:

$$\text{Error moviment: } \mathbf{e}_k(\mathbf{x}_{i_k-1}, \mathbf{x}_{i_k}) = f_{i_k}(\mathbf{x}_{i_k-1}, \mathbf{u}_{i_k}) - \mathbf{x}_{i_k} \tag{2.15}$$

$$\text{Error landmark: } \mathbf{e}_k(\mathbf{x}_{i_k}, \mathbf{l}_{j_k}) = h_k(\mathbf{x}_{i_k}, \mathbf{l}_{i_k}) - \mathbf{z}_k \tag{2.16}$$

Seguint el criteri establert anteriorment de no fer distinció entre el tipus d'observació, l'error entre dos estats a partir d'una observació es pot considerar de forma general:

$$\mathbf{e}_k(\mathbf{x}_{i_k-1}, \mathbf{x}_{i_k}, \mathbf{z}_k)$$

i els factors admeten la forma:

$$\phi_k = \exp(-0.5\mathbf{e}_k^T \Omega_k \mathbf{e}_k) \tag{2.17}$$

Per a cada factor es calcula l'error i, aleshores, la probabilitat conjunta es pot escriure com a producte de factors:

$$P(\mathbf{x}, \mathbf{z}) \propto \prod_{k=1}^{K} \phi_k \tag{2.18}$$

Maximitzar aquesta probabilitat és equivalent a minimitzar el seu *non-negative likelihood*, i solucionant l'equació al final es busca el resultat:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{k=1}^{K} \mathbf{e}_k(\mathbf{x}_i, \mathbf{x}_j)^T \Omega_k \mathbf{e}_k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.19}$$

on per simplificar la notació s'ha expressat els índexs i_k i j_k com a i i j, ja que cada factor $\mathbf{k}$ té els seus propis índexs associats.

Aquesta equació 2.19 és un problema no lineal de mínims quadrats que es pot entendre com minimitzar la suma de tots els errors entre les observacions fetes pels sensors i l'esperança d'aquella observació segons els models d'observació i l'estat que es té actualment. És un problema que es pot solucionar seguint diferents mètodes com podria ser el mètode de Gauss Newton, i es pot aprofitar l'estructura *sparse* del problema en el moment de calcular els jacobians per millorar-ne l'eficiència. Més detall sobre els mètodes de solucionar-ho es pot trobar a [3] i [6].

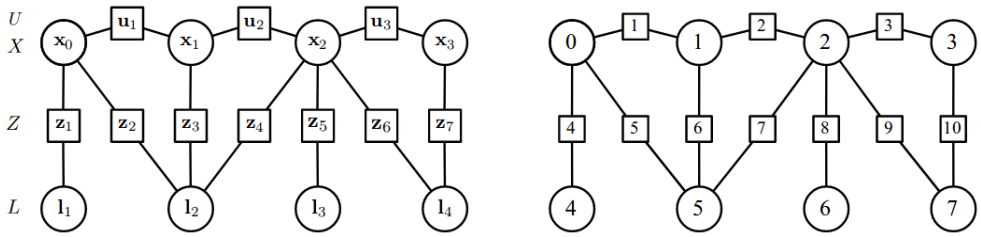

Figura 2.4: Factor graph pel problema d'SLAM de la Figura 2.2. Hi ha dos tipus de nodes, els cercles que són els estats que s'han d'estimar, i els quadrats, que són els factors associats a les observacions. A la figura de la dreta es pot veure com tots els estats es poden tractar indistintament, igual que tots els factors. Font: [3]

2.4 Solució al problema

Per solucionar problemes d'estimació en robòtica hi ha aproximacions diverses, però cada cop més hi ha la tendència d'explotar els conceptes de la teoria de Lie. En aquest apartat es fa un breu resum de com aplicar-ho en algoritmes d'estimació d'estats formulats com a factor graph basat en el que s'explica a [6].

Com s'ha comentat en la secció 2.3, amb el graph del problema amb els estats i els factors constituït, l'optimitzador ha de solucionar un problema de mínims quadrats de forma iterativa. Per tal de simplificar el problema que es mostra a la Figura 2.4, es considerarà el problema de la Figura 2.5 on es tenen 3 poses del robot en la trajectòria (de $\boldsymbol{x}_1$ a $\boldsymbol{x}_3$) i tres landmarks (de $\mathbf{b}_4$ a $\mathbf{b}_6$).

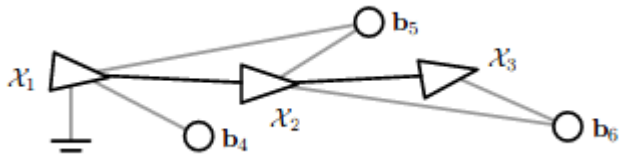

Figura 2.5: Problema de graph SLAM simplificat. Només es tenen 3 poses del robot $\boldsymbol{x}_i$ i 3 landmarks b_k. EL factor $\boldsymbol{x}_1$ estableix un prior per aconseguir observabilitat global. En negre es tenen dos factors de moviment i en gris cinc factors d'observacions a landmarks. Font: [6]

Així doncs, el vector d'estats de l'expressió 2.13 passa a quedar en la notació:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{b}_4 & \boldsymbol{b}_5 & \boldsymbol{b}_6 \end{bmatrix}^T \tag{2.20}$$

La secció 2.3 s'ha acabat amb l'expressió a resoldre 2.19. Aquesta expressió es pot escriure de la forma:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{k=1}^{K} \mathbf{e}_k(\mathbf{x})^T \Omega_k \mathbf{e}_k(\mathbf{x}) \tag{2.21}$$

i considerant

$$\mathbf{r} = \Omega^{\frac{T}{2}} \mathbf{e}(\mathbf{x}) \qquad \Omega = \Omega^{\frac{1}{2}} \Omega^{\frac{T}{2}} \tag{2.22}$$

s'arriba a l'expressió

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{k=1}^{K} \mathbf{r}_k(\mathbf{x})^T \mathbf{r}_k(\mathbf{x}) \tag{2.23}$$

Aquesta expressió, considerant la formulació de la Figura 2.5, es pot escriure tenint en compte que per cada factor $\mathbf{k}$ i ha una parella p que relaciona els índexs $i \in (1,2,3)$ i $k \in (4,5,6)$ en el conjunt $\mathcal{P} = \langle 1, 12, 23, 14, 15, 25, 26, 36 \rangle$ de parelles de nodes de cada observació:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{p \in \mathcal{P}} \mathbf{r}_p(\mathbf{x})^T \mathbf{r}_p(\mathbf{x}) \tag{2.24}$$

Això és la formulació del problema, però per resoldre'l cal fer-ho de forma iterativa partint d'un estat i actualitzant-lo progressivament per arribar a una solució. A l'opti-

mitzador a cada iteració es fa:

$$\mathbf{x} \longleftarrow \mathbf{x} \oplus \delta\mathbf{x}^* \tag{2.25}$$

on l'actualització òptima de l'estat surt de minimitzar:

$$\delta\mathbf{x}^* = \arg\min_{\delta\mathbf{x}} \sum_{p\in\mathcal{P}} \mathbf{r}_p(\mathbf{x}\oplus\delta\mathbf{x})^\top \mathbf{r}_p(\mathbf{x}\oplus\delta\mathbf{x}) \tag{2.26}$$

Per solucionar el problema iterativament, cada residu de la suma 2.4 es linealitza amb $\mathbf{r}_p(\mathbf{x}\oplus\delta\mathbf{x}) \approx \mathbf{r}_p(\mathbf{x}) + \mathbf{J}_{\mathbf{x}}^{\mathbf{r}_p}\delta\mathbf{x}$, on $\mathbf{J}_{\mathbf{x}}^{\mathbf{r}_p}$ són Jacobians sparse. Els blocs que són no nuls d'aquests Jacobians, que són $\mathbf{J}_{\boldsymbol{x}_i}^{\mathbf{r}_{ij}}$, $\mathbf{J}_{\boldsymbol{x}_j}^{\mathbf{r}_{ij}}$, $\mathbf{J}_{\boldsymbol{x}_i}^{\mathbf{r}_{ik}}$ i $\mathbf{J}_{\mathbf{b}_k}^{\mathbf{r}_{ik}}$, poden ser calculats de manera eficient seguint els mètodes explicats a [6]. Construint la matriu Jacobiana global i el vector de residus:

$$\mathbf{J} = \begin{bmatrix}
\mathbf{J}_{\boldsymbol{x}_1}^{\mathbf{r}_1} & 0 & 0 & 0 & 0 & 0 \\
\mathbf{J}_{\boldsymbol{x}_1}^{\mathbf{r}_{12}} & \mathbf{J}_{\boldsymbol{x}_2}^{\mathbf{r}_{12}} & 0 & 0 & 0 & 0 \\
0 & \mathbf{J}_{\boldsymbol{x}_2}^{\mathbf{r}_{23}} & \mathbf{J}_{\boldsymbol{x}_3}^{\mathbf{r}_{23}} & 0 & 0 & 0 \\
\mathbf{J}_{\boldsymbol{x}_1}^{\mathbf{r}_{14}} & 0 & 0 & \mathbf{J}_{b_4}^{\mathbf{r}_{14}} & 0 & 0 \\
\mathbf{J}_{\boldsymbol{x}_1}^{\mathbf{r}_{15}} & 0 & 0 & 0 & \mathbf{J}_{b_5}^{\mathbf{r}_{15}} & 0 \\
0 & \mathbf{J}_{\boldsymbol{x}_2}^{\mathbf{r}_{25}} & 0 & 0 & \mathbf{J}_{b_5}^{\mathbf{r}_{25}} & 0 \\
0 & \mathbf{J}_{\boldsymbol{x}_2}^{\mathbf{r}_{26}} & 0 & 0 & 0 & \mathbf{J}_{b_6}^{\mathbf{r}_{26}} \\
0 & 0 & \mathbf{J}_{\boldsymbol{x}_2}^{\mathbf{r}_{36}} & 0 & 0 & \mathbf{J}_{b_6}^{\mathbf{r}_{36}}
\end{bmatrix} \quad \mathbf{r} = \begin{bmatrix}
\mathbf{r}_1 \\ \mathbf{r}_{12} \\ \mathbf{r}_{23} \\ \mathbf{r}_{14} \\ \mathbf{r}_{15} \\ \mathbf{r}_{25} \\ \mathbf{r}_{26} \\ \mathbf{r}_{36}
\end{bmatrix} \tag{2.27}$$

l'expressió linealitzada 2.4 es transforma en el problema de minimitzar:

$$\delta\mathbf{x}^* = \arg\min_{\delta\mathbf{x}} \|\mathbf{r} + \mathbf{J}\delta\mathbf{x}\|^2 \tag{2.28}$$

Això es soluciona a partir de mínims quadrats utilitzant la pseudoinversa de $\mathbf{J}$, aconseguint el pas òptim $\delta\mathbf{x}^*$ utilitzat per actualitzar l'estat.

$$\delta\mathbf{x}^* = -(\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\mathbf{r} \tag{2.29}$$

$$\mathbf{x} \longleftarrow \mathbf{x} \oplus \delta\mathbf{x}^* \tag{2.30}$$

Aquest procés es fa de forma iterativa fins que es convergeix a la solució.

Comentari sobre la notació

L'operador $\oplus$ utilitzat en les expressions anteriors és una composició en l'àmbit de la Teoria de Lie.

$$\mathbf{x} \oplus \delta\mathbf{x} = \mathbf{x}\exp(\delta\mathbf{x}) \tag{2.31}$$

En el cas d'espais vectorials, utilitzat en el cas que ens ocupa per a la posició dels estats, $\oplus$ equival a $'+'$.

$$\mathbf{x} \oplus \delta\mathbf{x} = \mathbf{x} + \delta\mathbf{x} \tag{2.32}$$

En el cas dels quaternions, utilitzats per les orientacions dels estats, es té:

$$\mathbf{q} \oplus \delta\boldsymbol{\theta} = \mathbf{q}\exp(\delta\boldsymbol{\theta}) \tag{2.33}$$

on

$$\exp(\delta\boldsymbol{\theta}) = \begin{bmatrix} \cos\frac{\delta\theta}{2} \\ u\sin\frac{\delta\theta}{2} \end{bmatrix} \tag{2.34}$$

$$u = \frac{\delta\boldsymbol{\theta}}{\delta\theta} \tag{2.35}$$

$$\delta\theta = \|\delta\boldsymbol{\theta}\| \tag{2.36}$$

En aquest àmbit, la jacobiana $\mathbf{J}$ es calcula així:

$$\mathbf{J} = \lim_{\tau\to 0} \frac{f(\mathbf{x}\oplus\tau)\ominus f(\mathbf{x})}{\tau} \tag{2.37}$$

on $\ominus$ és l'invers de $\oplus$, és a dir:

$$Y \ominus X = \log(X^{-1}Y) \tag{2.38}$$

Capítol 3

Tríada Detector-Descriptor-Matcher

3.1 Introducció

Per a nosaltres, els humans, la visió és una capacitat natural i ens sembla relativament fàcil reconèixer formes o colors en una imatge, localitzar-nos en un entorn, seguir objectes en moviment, evitar obstacles i distingir objectes que ja havíem vist abans. Tot aquest seguit d'accions es duen a terme gràcies a un sistema visual evolucionat que portem entrenant des de que hi comencem a veure. Això que ens resulta relativament tant fàcil, per a un ordinador no és gens trivial, ja que on nosaltres hi veiem una imatge, per una màquina no és més que un munt de números sense sentit. La visió per computador és la disciplina que, des de ja fa anys, s'encarrega de processar les imatges per extreure'n informació i de fet s'han dedicat molts esforços, amb més o menys èxit, a intentar implementar les accions que abans comentava. No es tracta només de plasmar el món en una imatge, es tracta de fer que l'ordinador hi pugui 'veure', amb tot el que això comporta.

En aquest sentit, en visió per computador s'han fet moltes aproximacions a tècniques diverses per tal d'extreure informació d'aquest 'conjunt de punts' que és una imatge, i aquest projecte es centrarà en treballar amb les unitats més petites d'informació que se'n poden extreure anomenades *features*.

Un *feature* és un punt en una imatge que per si sol no és informatiu, però que pot ser explotat per tal d'utilitzar les imatge pel que interessa en el projecte, que és fer tracking, ja que un *feature* en una imatge 2D està lligat a un *landmark* en l'espai 3D.

Per tant, un *feature* és un punt en una imatge que presenta una sèrie de característiques interessants:

- Són punts que destaquen, que són fàcils de distingir del seu entorn, i que per tant es poden **detectar**. Un exemple en seria una cantonada d'un rectangle.

- Són punts diferents de la resta, es poden **descriure** a partir del seu entorn local de manera que es poden diferenciar d'un altre *feature*.

– Són punts que, en destacar i ser únics, es poden trobar en altres imatges, és a dir, es pot fer **match** de *features* que corresponen al mateix element tridimensional.

Com es pot extreure d'aquestes característiques, sense entendre què significa la imatge es poden fer servir els *features* de la imatge per fer track en diferents imatges d'algun *landmark*. En les següents seccions, s'exposa com s'ha treballat en aquest projecte en aquests tres processos tant associats a un *feature*: detectar, descriure i fer match.

3.2 OpenCV i vision_utils

Amb l'objectiu de fer una anàlisi dels diferents *detectors*, *descriptors* i *matchers*, s'han utilitzat eines que implementen algoritmes d'aquest tipus per seleccionar una combinació útil per al projecte. Com que aquest projecte es desenvolupa en el marc de la llibreria en C++ Wolf creada a l'IRI (Institut de Robòtica i Informàtica Industrial), s'ha fet servir una eina associada a aquesta que s'anomena vision_utils. Wolf té una política de treball que fa que per utilitzar-lo amb llibreries externes es crea un *wrapper* de la llibreria externa per tenir una API útil per a Wolf amb les funcionalitats que es necessiten. En aquest sentit, vision_utils fa aquesta funció per les eines relacionades amb visió per computador i és un *wrapper* de la llibreria OpenCV en una interfície més útil per treballar.

Així doncs, la base de visió per computador dels algoritmes de detectors, descriptors i matchers es troba implementada a OpenCV.

OpenCV és una llibreria *open source* de visió per computador que ofereix una gran quantitat de possibilitats i està estructurada de forma modular. Per la part que interessa, a banda dels mòduls bàsics per gestionar imatge i vídeo, es farà ús del mòdul *Features2D* que és on estan implementats els mètodes *detect-descript-match* de la llibreria.

Els conceptes d'OpenCV i dels mètodes utilitzats s'han extret en particular de [7] i de la documentació d'OpenCV, on es pot trobar l'herència de classes per als mètodes detect-descript-match, l'API de la llibreria i les referències al *papers* de cada mètode implementat.

3.3 Detectors

Els *detectors* són els mètodes destinats a trobar punts que destaquen en una imatge. A OpenCV, aquests *features* es representen amb un objecte anomenat cv::KeyPoint. Els atributs que emmagatzema aquest objecte donen una idea molt clara del que es té en compte en determinar un *feature*:

– **(x,y)**: coordenades del punt 2D, ja que un feature correspon a un píxel concret.

– **Size**: diàmetre del veïnat significatiu. Aquest concepte ja dóna la idea que un keypoint és un element característic que destaca respecte l'entorn local que l'envolta.

– **Response**: és un valor que fa referència a la fortalesa del keypoint. Sovint en un mateix veïnat hi ha keypoints molt propers, aquest valor pot servir per classificar-los i escollir els més estables.

– **Angle**: per a un keypoint hi ha detectors que calculen una orientació per aconseguir invariància a l'escala.

– **Octave**: hi ha keypoints que es detecten a múltiples escales per aconseguir invariància a l'escala.

Per obtenir aquests paràmetres, s'han considerat tots els mètodes possibles d'OpenCV que estan implementats a vision_utils. Dels mètodes existents, n'hi ha una gran part que existeixen com a detector i descriptor conjuntament, ja que en ser dues tasques tant unides es desenvolupen al mateix temps, però es pot utilitzar qualsevol part per separat.

Detector	Detector-Descriptor
AGAST [8]	SIFT [9]
FAST [10]	SURF [11]
GFTT (Shi-Tomasi) [12]	KAZE [13]
HARRIS [14]	AKAZE [15]
MSER [16]	BRISK [17]
SBD (Simple Blob Detector)	ORB [18]

Taula 3.1: Mètodes per detectar keypoints

3.4 Descriptors

Els *descriptors* són els mètodes destinats a descriure els keypoints detectats en una imatge, i per 'descriure' s'entén calcular alguna mètrica per al keypoint que el permeti ser diferenciat d'un altre.

Els mètodes implementats a vision_utils que s'han considerat es veuen a la Taula 3.2.

Detector-Descriptor	Descriptor
SIFT	BRIEF [19]
SURF	DAISY [20]
KAZE	FREAK [21]
AKAZE	LATCH [22]
BRISK	LUCID [23]
ORB	

Taula 3.2: Mètodes per descriure keypoints

Cada mètode emmagatzema un descriptor de tamany diferent segons la quantitat d'informació que es vulgui guardar. En general, es pot diferenciar entre descriptors que es basen en calcular el gradient i que són més costosos en còmput i memòria (com podria ser SIFT o SURF), i entre descriptors binaris (com BRIEF), que es basen en comparacions de dos píxels i que permeten més velocitat i descriptors més petits. A més, també es pot reduir el temps de còmput si no es descriu el punt amb invariància a la rotació o en l'espai d'escales.

3.5 Matchers

Els *matchers* són el tercer element de la tríada i s'encarreguen d'associar un keypoint en una imatge amb un altre en una imatge diferent. Bàsicament es basen en comparar els descriptors per determinar si són el mateix punt.

A OpenCV, els matchers estan implementats de manera diferent segons si l'objectiu és fer reconeixement d'objectes a partir d'una base de dades o, en el cas d'aquest projecte, fer tracking. En el cas de fer tracking, s'ha de tenir una llista de descriptors *query* i una llista de descriptors *train* corresponents a dues imatges i l'algoritme s'encarrega de comparar-los i retornar els millors matches en un objecte anomenat cv::DMatch que conté l'índex dels keypoints *query* i *train* i un valor que és la distància de match (que es pot associar a la qualitat).

El fet de definir quin és el 'best' match és feina del mètode que es faci servir. A la Taula 3.3 es poden visualitzar les dues opcions de matcher que hi ha. Per una banda, BRUTEFORCE es dedica a comparar per al descriptor *query* cadascun dels descriptors *train* un per un i selecciona el millor.

Per altra banda, FLANNBASED utilitza mètodes de *nearest neighbor search* per obtenir el millor match i acostuma a ser, per tant, un mètode més ràpid.

Matchers
BRUTEFORCE
FLANNBASED [24]

Taula 3.3: Mètodes per fer match de keypoints

La comparació dels descriptors, en el cas de FLANNBASED per defecte s'acostuma a fer en norma L2, però en el cas de BRUTEFORCE es poden comparar els descriptors de múltiples maneres (veure Taula 3.4). La distància de Hamming, que actua com una porta XOR i compara cada bit un a un del descriptor i compta quants n'hi ha de diferents, només es pot utilitzar en descriptors binaris.

Matcher	Mètode de comparació
BRUTEFORCE	norma L2
BRUTEFORCE_L1	norma L1
BRUTEFORCE_HAMMING	distància de Hamming

Taula 3.4: Mètodes per fer match de keypoints

Per últim, cal comentar que en fer match no només hi ha la possibilitat d'obtenir un match per keypoint, es pot fer servir un mètode anomenat knnMatch que proporciona els 'n' millors matches, o radiusMatch, que et dona els millors matches dins un radi determinat.

A vision_utils hi ha implementades funcionalitats extres un cop s'ha fet match. Els mètodes d'OpenCV proporcionen els 'millor' match però aquest pot ser erroni i per això un cop s'ha fet el match s'apliquen filtres al resultat per tal d'eliminar els *outliers* fent comprovacions extres i tenint en compte la distància dels matches.

3.6 Anàlisi de la tríada

Per tal de fer una anàlisi de la millor tríada, s'ha dissenyat un algoritme que utilitza els *detectors*, *descriptors* i *matchers* d'OpenCV encapsulats a través de vision_utils. L'objectiu del projecte és partir de la seqüència d'imatges obtinguda amb una càmera i fer track a través de les imatges de *features* associats a un *landmark* exterior. El primer pas és la detecció, descripció i match amb OpenCV, i per avaluar el funcionament dels mètodes disponibles s'ha dissenyat un algoritme per extreure mètriques que permetin comparar les tríades.

De cada tríada de detectors-descriptor-matcher, hi ha una sèrie de paràmetres associats

a cada mètode per tal d'ajustar el seu funcionament. Tots els paràmetres de cada detector i descriptor es troben detallats en taules a l'annex A.

3.6.1 Algoritme

El procediment dissenyat segueix els passos que es veuen a l'algoritme 1. Bàsicament, aquest algoritme itera sobre les diferents combinacions de *detector*, *descriptor* i *matcher* i, per cada una d'elles, analitza una seqüència de vídeo i n'extreu un conjunt de mètriques que s'emmagatzemen per una anàlisi posterior.

Dins de l'algoritme, cal destacar el disseny que s'ha fet per obtenir mètriques dels tracks. Amb OpenCV, el que es pot fer utilitzant un mètode de matching és fer match de *features* d'una imatge (*query*) contra l'altra (*train*). Tal i com s'ha dissenyat l'algoritme, cada cop que es fa match es realitza dels descriptors de la imatge antiga (*query*) contra els descriptors de la imatge més actual (*train*) [1]. Això s'ha decidit així perquè en fer match, per cada punt *query* es busca el millor punt de *train*, i pot passar que un punt *train* sigui el millor per a múltiples punts *query*. L'elecció de quina imatge és *query* i quina es *train*, per tant, pot comportar que els tracks es ramifiquin o que els tracks s'ajuntin, i per comoditat en guardar els tracks i calcular les mètriques s'ha escollit que *query* fos la informació del passat per tal que els tracks es vagin ajuntant.

Fins on arriba OpenCV només es pot fer track entre dues imatges, i per emmagatzemar la història de tracks s'ha utilitzat un mapa de la llibreria estàndard de C++. Per cada iteració de l'algoritme sobre cada imatge, es tenen disponibles els KeyPoints corresponents a la imatge *query* (anterior), i els KeyPoints corresponents a la imatge *train* (actual). El disseny s'ha fet tal que per accedir al mapa la clau és sempre l'índex d'un KeyPoint corresponent *train*, i l'element que es guarda amb aquesta clau és un vector amb les coordenades dels punts corresponents al KeyPoints del track.

D'aquesta manera, cada cop que s'actualitza l'algoritme i la imatge que era *train* passa a ser *query*, es pot saber si la nova imatge *train* té algun match que ja existia en el passat mirant si l'índex del match que ha fet amb *query* és una clau del mapa. En cas de que ja existís, s'allarga la longitud del track, i si no hi és se'n crea un de nou. Per disseny de l'algoritme també s'ha decidit només mantenir vius els tracks que estan presents a *train*, i anar eliminant la història de tracks que ja no té continuïtat al moment actual.

[1] Aquesta nomenclatura pot semblar confusa ja que normalment a OpenCV *query* és la informació nova i *train* és la informació que ja es tenia. Si està fet així és perquè s'ha dissenyat per tal que es faci match des de la informació antiga cap a la nova, i per tant els noms estan invertits.

Algorithm 1: Anàlisi tríada detector-descriptor-matcher

```
Declarar les opcions disponibles de det-des-mat;
Inicialitzar fitxer de text;
for cada opció de detector do
    for cada opció de descriptor do
        for cada opció de matcher do
            Crear detector, descriptor i matcher amb els paràmetres corresponents;
            Inicialitzar la captura d'imatges;
            Inicialitzar mètriques;
            while hi hagi imatges, per cada imatge do
                Detectar KeyPoints;
                Calcular Descriptors;
                Fer match;
                Omplir el mapa de tracks;
                Visualitzar els tracks i els KeyPoints;
                Updates;
            end
            Calcular mètriques;
            Escriure mètriques a un fitxer de text;
        end
    end
end
```

3.6.2 Criteris a considerar per a la tríada

Des del bon començament, quan s'han estudiat les diferents possibilitats de mètodes de detecció-descripció-match, s'ha tingut clar que la millora alternativa depèn dels objectius per als quals es vol seleccionar, ja que cada mètode està dissenyat per oferir una bona resposta en les prestacions que pretén emfatitzar, cosa que fa que perdi qualitat en altres aspectes. Per tant cal decidir les especificacions que es busquen i fer-ne un balanç per escollir l'alternativa.

Les prestacions que s'han considerat a tenir en compte per als objectius del projecte són:

- **Rapidesa**: El procés de *detect*, *descript* i *match* ha de ser el més ràpid possible. Les imatges arriben aproximadament entre 30 i 40 ms, i si es vol processar a temps real no pot superar aquest valor.

- **Longitud dels tracks**: La tríada ha de permetre establir tracks el més llarg possibles per no haver d'estar constantment buscant nous tracks i tenir la història del

passat recollida en el track.

- **Invariabilitat a l'escala**: La tríada ha de poder detectar un mateix *feature* a diferents escales en l'espai i poder reconèixer que és el mateix.

Especificació	Mètrica	Rang de valors
Rapidesa	temps t (ms)	t < 40 ms
Longitud dels tracks	Número de *features* per track N	N>2
Invariabilitat a l'escala	Número d'octaves O	$0 \leq O \leq 2$

Taula 3.5: Taula d'especificacions

Fent un raonament previ sobre el propòsit del projecte, s'ha observat que el que es busca és fer track entre imatges consecutives i que, per tant, entre una imatge i la imatge immediatament següent ha passat un temps molt curt i, per tant, és factible fer l'assumpció de que la imatge no haurà canviat gaire. Això posa en dubte la importància de la invariabilitat en l'escala de la tríada, i també a permès descartar la següent opció:

- **Invariabilitat a la rotació**: No es considera ja que en dues imatges consecutives es fa la hipòtesi que no canviarà significativament el descriptor d'un mateix KeyPoint degut a la rotació.

3.6.3 Mètriques seleccionades

Per tal d'analitzar la tríada que compleixi amb les especificacions del projecte, s'han seleccionat una sèrie de mètriques que s'han mesurat a l'algoritme 1.

Detector	Descriptor	Matcher	Tracks
Núm. KeyPoints	$\bar{t}^{DES}_{imatge}$ (ms)	$\bar{t}^{MAT}_{imatge}$	Màxima llargada l (frames)
Núm. Octaves	$s^{DES}_{t_{imatge}}(ms)$	$s^{MAT}_{t_{imatge}}(ms)$	Número d'ambigüitats
$\bar{t}^{DET}_{imatge}(ms)$	$\bar{t}^{DES}_{punt}$	$\bar{t}^{MAT}_{punt}$	$\bar{l}(frames)$
$s^{DET}_{t_{imatge}}(ms)$	$s^{DES}_{t_{punt}}(ms)$	$s^{MAT}_{t_{punt}}(ms)$	$s_l(frames)$
$\bar{t}^{DET}_{punt}$			Número de tracks totals
$s^{DET}_{t_{punt}}(ms)$			Número de tracks $l = 1$
			Número de tracks $l = 2$
			Número de tracks $l \geq 3$

Taula 3.6: Taula de mètriques

3.6.4 Resultats i selecció de la tríada

En començar a fer una anàlisi de les alternatives, es veu ràpidament que hi ha un munt de combinacions de detectors i descriptors i, per tant, en primer lloc s'ha fet una primera criba basada en l'experimentació inicial per descartar aquelles opcions que ja s'ha vist que no eren viables i poder tenir un problema més abordable. S'ha descartat utilitzar els detectors MSER i SBD, els detector-descriptor SIFT, SURF i KAZE i el descriptor DAISY. La justificació és l'elevat temps d'execució que comporten, que és molt superior als seus rivals.

Per altra banda, per reduir també el nombre de combinacions, s'ha fet una taula de totes les alternatives de detector-descriptor amb BRUTEFORCE i el mateix per FLANNBASED. La conclusió a la qual s'ha arribat és que FLANNBASED és molt més ràpid en fer el match que no pas BRUTEFORCE (o qualsevol de les seves variants amb distància de Hamming) independentment de l'opció detector-descriptor. El ratio del temps de match BRUTEFORCE/FLANNBASED té un valor de 3 a 8 en la seqüència de vídeo utilitzada, pel que s'ha descartat BRUTEFORCE.

Encara que es redueixi la combinació de tríades, el problema encara agafa més magnitud si es considera que cada detector, descriptor o matcher té una sèrie llarga de paràmetres associats que també es poden ajustar (veure annex A). Per tal d'evitar la dimensionalitat que s'adquireix si es volen fer totes les combinacions, s'ha decidit modificar els paràmetres de cada combinació per tal que tots els detectors detectin un nombre de punts a cada imatge del mateix ordre de magnitud, ja que és la millor manera de poder comparar els temps d'execució de cada combinació.

Per últim, abans de quedar-se amb el grup de combinacions en aquest primer filtratge, també s'ha fet una taula amb els temps i longitud dels tracks en treballar amb invariància d'escala i detectar i descriure a múltiples octaves. El resultat ha estat que com que entre una imatge i l'altra no hi ha gaire variació, detectar en múltiples octaves no aporta millores significatives, al contrari, alenteix molt el temps d'execució. Per tant, seguint el mateix criteri que s'havia considerat en les especificacions per la invariància en la rotació, també s'ha descartat la invariància en escala i s'ha fixat que tots els mètodes treballin a zero octaves, és a dir, sense modificar l'escala.

De la resta d'opcions, s'ha fet servir l'algoritme dissenyat per obtenir una taula definitiva de mètriques a comparar. Les opcions que s'han estudiat, feta la criba, són:

- **Detectors**: AKAZE, AGAST, FAST, ORB, BRISK
- **Descriptors**: AKAZE, ORB, BRIEF, BRISK, FREAK, LATCH, LUCID
- **Matchers**: FLANNBASED

D'aquestes opcions se'n poden extreure 31 combinacions possibles, ja que el descriptor

AKAZE només és compatible amb el seu detector. S'ha fet l'anàlisi de les opcions sobre una seqüència de vídeo de 157 frames en l'entorn del laboratori de robòtica mòbil de l'IRI.

Fent una anàlisi dels resultats de temps de detecció de cada detector, ja es pot observar en la Figura 3.1 que el detector AKAZE i el detector BRISK donen temps superiors a la freqüència d'arribada d'imatges, i tot i que això es pot ajustar reduint el número de KeyPoints per imatge, ja es veu que són significativament més lents que la resta.

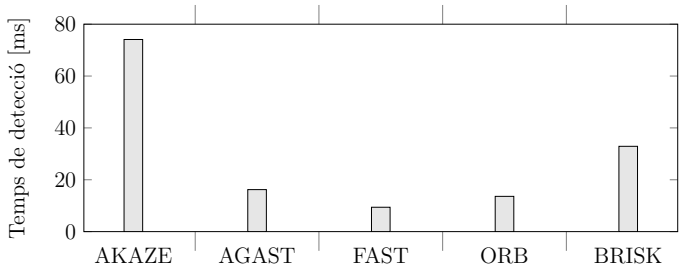

Figura 3.1: Temps de detecció per imatge de cada detector.

En la Figura 3.2 es mostra el temps de descripció per a cada detector combinat amb totes les possibles opcions de descriptor. Se'n pot extreure que el descriptor AKAZE (que només és compatible amb el detector AKAZE) no és viable i que el descriptor LATCH sempre dona resultats per sobre la resta de descriptors.

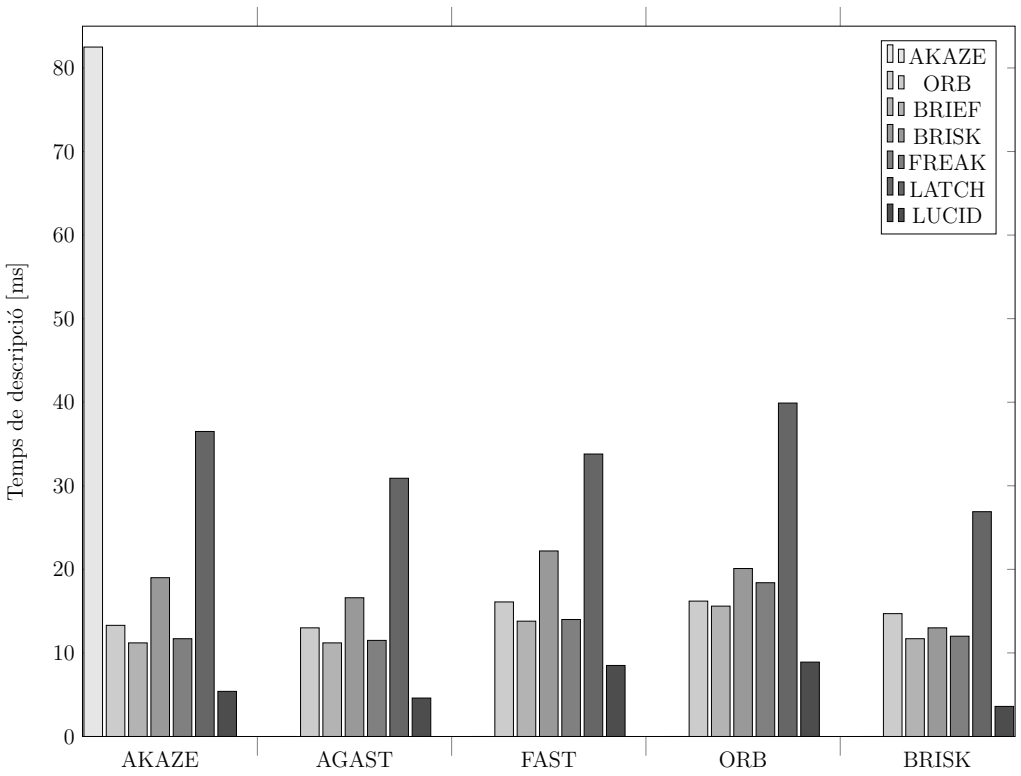

Figura 3.2: Temps de descripció per imatge de cada detector amb les múltiples combinacions de descriptors. El descriptor AKAZE només és compatible amb el detector AKAZE, per això en la resta de detectors de l'eix horitzontal la columna corresponent està a zero.

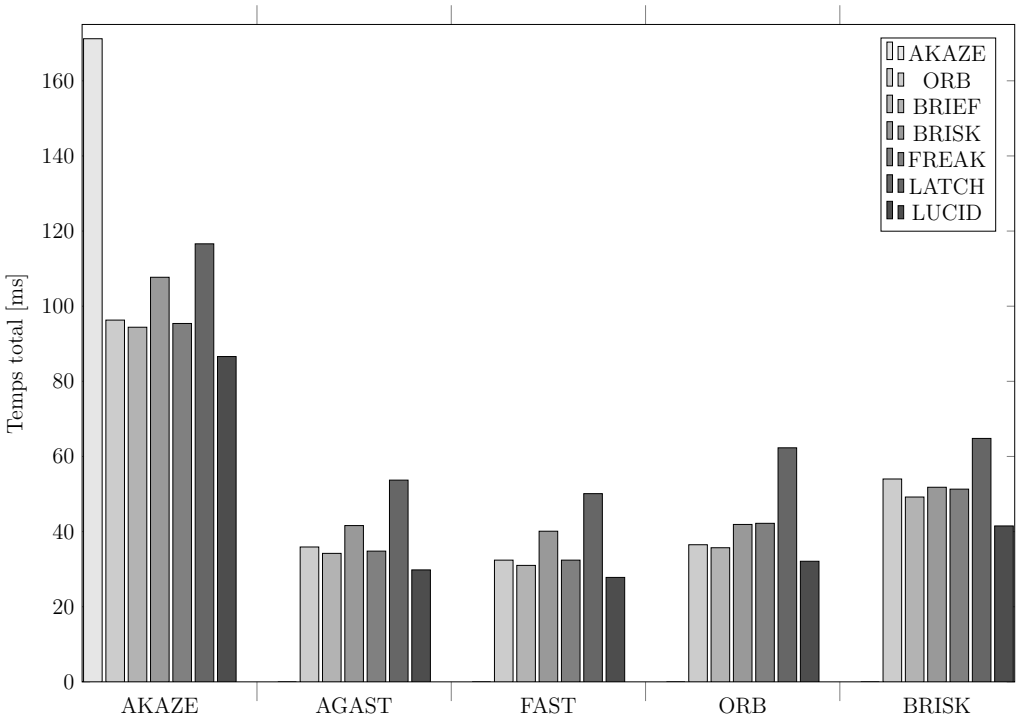

Figura 3.3: Temps total de detect-descript-match per imatge de cada detector amb les múltiples combinacions de descriptors utilitzant el matcher FLANNBASED. El descriptor AKAZE només és compatible amb el detector AKAZE, per això en la resta de detectors de l'eix horitzontal la columna corresponent està a zero.

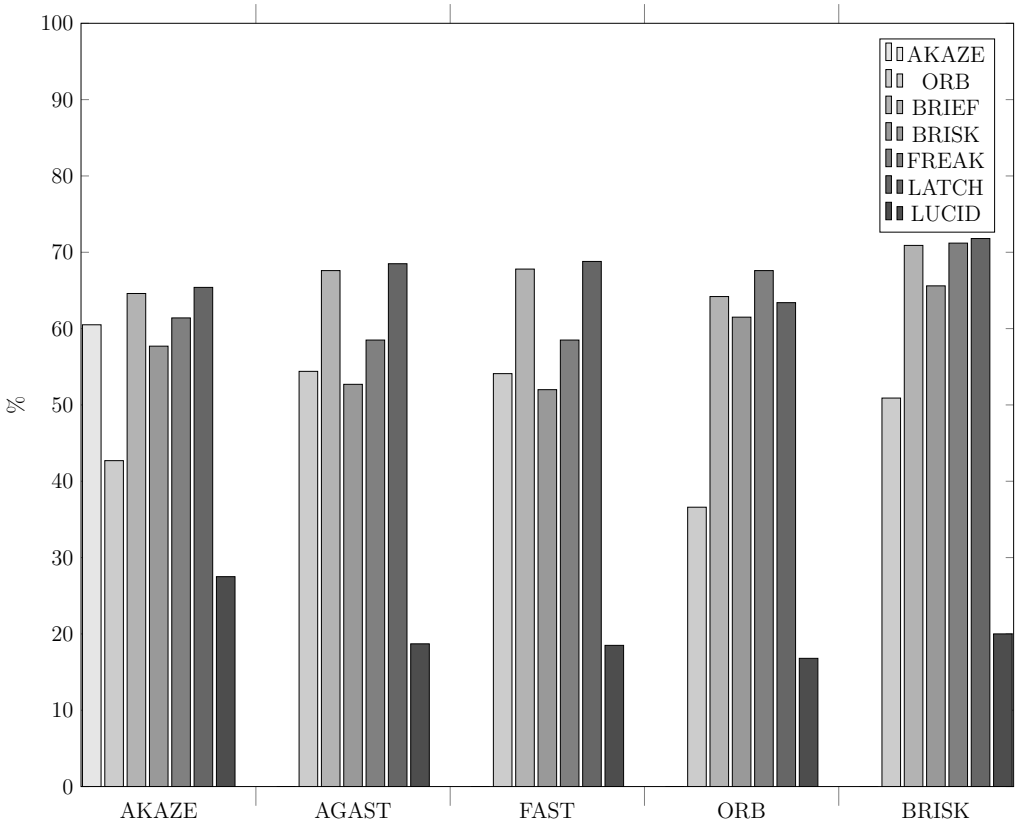

Figura 3.4: Percentatge de tracks útils totals ($l \geq 2$) que són de longitud $l \geq 3$.

Observant la Figura 3.4, es pot veure com de totes les opcions de descriptor, el LATCH és el que sempre dona un percentatge inferior de tracks útils sigui quin sigui el detector. Justament el LATCH era el que tenia temps de descripció més baix, i segurament, això és en detriment de la qualitat d'aquest descriptor de manera que el track es veu afectat.

Per altra banda, també s'han analitzat les longituds nominals dels tracks i s'ha vist que els que arriben a donar tracks de longitud més llarga són utilitzant AKAZE i BRISK com a detector, i de fet AKAZE dóna uns tracks de gran longitud i constants respecte les altres opcions. Justament, però, aquests són els que donen temps de detecció més alts, i segurament per això els tracks són més estables al llarg de diferents imatges.

Tríada seleccionada

Observant tots els raonaments anteriors, està clar que el problema de seleccionar una tríada és difícil d'abordar degut al gran nombre de possibilitats. Amb els resultats a la mà s'ha evidenciat que aquelles opcions que donen tracks més llargs ho fan a costa d'un temps de còmput més elevat, i que aquells massa ràpids perden qualitat de tracks. Tenint en compte les especificacions i considerant, tal i com s'ha comentat, que no es considera l'escala, les opcions que queden com a finalistes són AGAST, FAST i ORB com a detectors, BRIEF i ORB com a descriptors i FLANNBASED com a matcher. Aquests finalistes surten sobretot de considerar un paràmetre crític com és el temps d'execució, ja que es pretén utilitzar-ho en temps real.

D'aquestes opcions, és relativament fàcil escollir la guanyadora si es té en compte que ORB és un detector-descriptor que té implementat un FAST com a detector i un BRIEF com a descriptor. D'aquesta manera, realment només es tenen dues alternatives com a detector: FAST o AGAST, una alternativa com a descriptor que és BRIEF i el FLANNBASED com a matcher.

Entre AGAST i FAST la diferència és minsa perquè els dos estan implementats amb l'objectiu de prioritzar la velocitat, però FAST sembla funcionar lleugerament més ràpid.

En definitiva, tenint a FAST com a guanyador de detector i tenint tanta semblança de comportament entre les opcions guanyadores, la tria final és la combinació: ORB-ORB-FLANNBASED, ja que ORB implementa un FAST i un BRIEF i a més permet més versatilitat sempre que es necessiti ja que es poden ajustar més els paràmetres per assolir invariàncies (rotació o escala) encara que per aquest projecte en concret no es pretenguin utilitzar.

Detector	Descriptor	Matcher
ORB	ORB	FLANNBASED

Taula 3.7: Tríada seleccionada.

Capítol 4

WOLF – Windowed Localization Frames

En aquest capítol es veurà una introducció a l'estructura de Wolf basat en la feina feta a l'IRI, explicada en paraules pròpies. Es centrarà en els detalls que es necessiten al capítol 5 per implementar un algoritme de *bundle adjsutment* per fer localització i cartografia.

4.1 Introducció a WOLF: el Wolf Tree

Wolf és una llibreria per solucionar problemes de localització en robòtica mòbil, com ara *simultaneous localization and mapping* (SLAM) o odometria visual. Està sent desenvolupada per un grup d'investigadors dirigits per en Joan Solà al grup de robòtica mòbil a l'IRI – Institut de Robòtica i Informàtica Industrial.

Wolf és principalment una eina per organitzar i gestionar la informació involucrada en el problema de localització en robòtica. La seva estructura està pensada per permetre la coexistència de sensors de diferents tipus i construeix vectors d'estat a partir d'un conjunt de keyframes que defineixen la trajectòria del robot i altres estats com ara *landmarks* o paràmetres dels sensors per calibració.

La llibreria requereix diversos *front-ends*, un per cada sensor diferent, per gestionar l'adquisició de dades i construir l'estructura amb totes les dades organitzades. També requereix un *back-end* que està constituït bàsicament pel *solver*.

L'estructura bàsica de Wolf és un arbre de classes base reproduint el problema, tal i com es pot veure plasmat a la Figura 4.1. En aquesta estructura hi ha la classe base **problem** que està enllaçada a tres estructures de dades que contenen la informació per solucionar el problema:

- **Hardware**: *Com obtenir dades i com utilitzar-les pel nostre propòsit?*

  Classe base contenint totes les classes associades amb l'adquisició d'informació utilitzant sensors i el processament d'aquesta per omplir la resta de l'arbre.

- **Trajectory**: *On sóc?*

  Classe base contenint els frames del robot i la informació derivada de cada pose del robot, incloent les captures associades a cada frame, els *features* utilitzats en cada captura i els *factors* establerts per solucionar el problema.

- **Map**: *Com és el món?*

  Classe base que conté una llista dels *landmarks* observats en el món exterior.

L'estructura de Wolf proporciona la funcionalitat bàsica per gestionar tots els elements del problema. A partir d'això, utilitzant el polimorfisme de C++ es poden derivar les classes base per implementar funcionalitats més complexes adaptades als propòsits i necessitats de l'aplicació.

En l'arbre de Wolf cada fill està connectat al seu node pare a partir d'un *pointer* i cada node pare conté un *pointer* o una llista de *pointers* als seus fills. El que és interessant és que aquest esquema d'arbre es trenca amb alguns enllaços que augmenten la connectivitat de l'estructura i gràcies a aquestes associacions és equivalent al *factor graph* que es pot resoldre amb optimització no lineal.

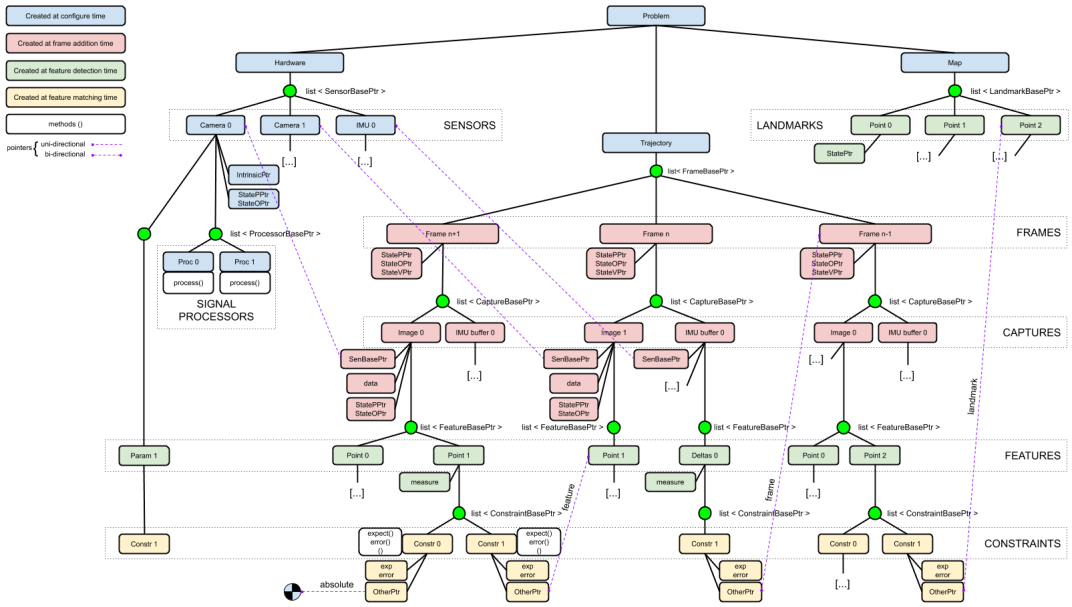

Figura 4.1: Wolf Tree

Estructura de plugins

La llibreria de Wolf està pensada per ser una eina general per processar dades de sensors i solucionar problemes de localització. Seguint aquest desig d'oferir versatilitat i adaptabilitat, Wolf es divideix en diverses parts anomenades plugins.

Core és el plugin principal que conté totes les funcionalitats bàsiques reflectides en l'arbre de Wolf (Figura 4.1).

Derivant classes base, es creen múltiples capes d'abstracció que creixen en complexitat i són més properes a cada problema específic. En aquest projecte, el processador bundle adjustment implementat és part del plugin **Vision** i està construït sobre el plugin **Core**.

L'estructura de plugins té cinc plugins diferents a part del **Core**. Alguns d'ells depenen d'altres plugins per funcionar, com es pot veure a la Figura 4.2.

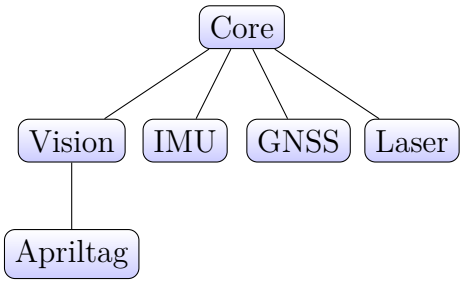

Figura 4.2: Estructura de plugins i dependència en altres plugins

4.2 Problem i el solver

El **Problem** és la classe que es troba a dalt de tot de l'arbre de Wolf i la seva funcionalitat principal és gestionar la informació de les classes inferiors en l'arbre. Conté un *pointer* per accedir a cadascuna de les tres branques immediatament inferiors en l'arbre: Hardware, Map i Trajectory.

Per fer servir la llibreria Wolf des de l'exterior, el primer pas és crear un objecte Problem segons quina estructura de frames es necessiti (PO 2D o 3D, o POV) depenent dels paràmetres que es vulguin estimar (posició, orientació o velocitat). A partir d'aquest objecte es pot accedir a les tres branques principals per obtenir informació, instal·lar sensors i processadors, i gestionar informació necessària pel solver.

Solver

El solver que es fa servir a Wolf per trobar una solució òptima al problema és Ceres (tot i que hi ha la intenció de poder afegir solvers diferents en un futur). En la filosofia de Wolf, per fer servir llibreries externes el que cal fer és crear un *wrapper* que permeti comunicar Wolf amb l'API de l'altra llibreria. Així doncs, a Wolf hi ha implementat un *wrapper* per

a Ceres que s'encarrega d'accedir a l'arbre i obtenir la informació necessària com els state blocks o els factors per poder solucionar el problema. D'aquesta manera Wolf funciona de forma independent al solver i es pot tenir un *front-end* que capta, processa i gestiona la informació i un *back-end* que optimitza els estats.

4.3 Branca Hardware

La branca del hardware proporciona una estructura que conté tots els sensors del robot utilitzats per obtenir mesures, amb tots els paràmetres extrínsecs i intrínsecs. També conté tots els processadors que implementen els algoritmes que utilitzen les dades que arriben dels sensors per omplir l'arbre i crear els enllaços necessaris per tenir tota l'estructura del problema definida.

La classe **HardwareBase** conté una llista de sensors que poden ser de diferents tipus, ja siguin exteroceptors com la visió o propioceptors com una IMU. Pot afegir nous sensors a la llista cada cop que un nou sensor és instal·lat.

4.3.1 Sensor

La classe **SensorBase** conté un enllaç al seu pare, la classe **HardwareBase**, i una llista de *pointers* a classes **ProcessorBase** que poden ser derivades per desenvolupar una tasca concreta.

També guarda tota la informació necessària sobre el sensor, que bàsicament és:

- Paràmetres extrínsecs (posició i orientació)
- Paràmetres intrínsecs
- Soroll del sensor i covariància del soroll

Pel que fa als mètodes a destacar de la classe base, a part dels mètodes per accedir als atributs de la classe que utilitzen altres classes derivades o externes, hi ha un mètode per afegir processadors a un sensor i un mètode per processar les dades que arriben al sensor que comunica directament amb un processador.

Particularització per al projecte

La classe SensorBase té algunes classes derivades implementant diferents tipus de sensors compatibles amb Wolf. Per aquest projecte es necessita la classe derivada que implementa una càmera per obtenir imatges de l'exterior (veure Figura 4.3).

Aquesta classe ja està implementada en el plugin Vision i s'anomena **SensorCamera**.

Té la funcionalitat de la classe SensorBase i afegeix atributs i mètodes particulars que necessita la càmera:

- Amplitud i altura de la imatge
- Paràmetres intrínsecs del model de càmera pinhole

- Coeficients de distorsió radial

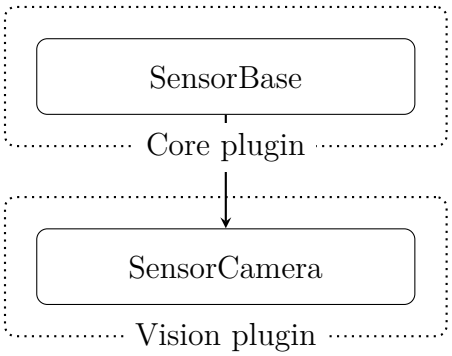

Figura 4.3: Classes derivades del sensor per al projecte

4.3.2 Processador

El processador és l'element de l'arbre de Wolf dedicat a omplir tots els altres elements de l'arbre, per tant és un dels elements de més complexitat i el que ha de proporcionar la funcionalitat que necessita el projecte.

A la Figura 4.4 es pot veure l'estructura jeràrquica del processador que es desenvolupa en aquest projecte. Hi ha quatre capes d'abstracció, cadascuna afegint més funcionalitats al processador.

La classe **ProcessorBase** és la classe base per a tots els processadors. En el capítol 5 s'explicarà millor l'estructura de capes d'aquest processador per entendre com s'ha implementat el processador Bundle Adjustment.

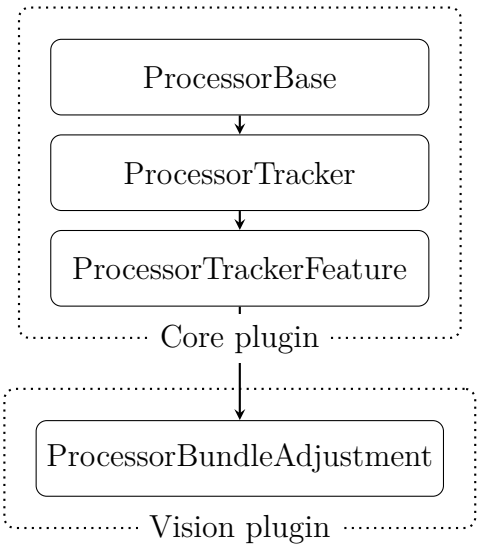

Figura 4.4: Classes derivades del processador per al projecte

4.4 Branca Trajectory

La branca de la trajectòria conté tota la informació referent al moviment del robot i al seu estat al llarg de la trajectòria que descriu. És una estructura que té diferents capes d'organització per tal de tenir organitzada tota la informació per calcular els factors que permetran resoldre el problema.

Les funcions principal a destacar són els mètodes per accedir als frames, que permeten que des de l'arbre de wolf es pugin obtenir *pointers* a l'últim frame del robot o a la llista de frames.

4.4.1 Frame

El primer element que es troba en la branca és la classe **FrameBase**. La trajectòria té una llista de frames que corresponen a les diferents poses del robot al llarg del moviment.

Cada Frame del robot es pot classificar segons sigui **KEY_FRAME** o **NON_KEY_FRAME**. Els frames classificats com a KEY són aquells essencials en la trajectòria del robot i es classifiquen amb aquesta categoria segons el criteri que estableix el processador que s'utilitza per solucionar el problema, com podria ser fer un KEY frame a cada imatge que s'obté, fer-ne cada cert interval de temps de forma regular, o cada cop que es compleix una condició com que s'han perdut *tracks* en les seqüències de dades que arriben al processador. En la llista de la trajectòria, només s'hi emmagatzemen els KEY frames, que són els que donen informació rellevant per reconstruir la trajectòria, i l'últim Frame del robot que es va substituint per cada nou Frame del robot si aquest no és KEY.

Com a element a destacar de la classe el més important és el que s'anomena vector d'**State Blocks**. En l'estructura de Wolf, els estats a estimar s'emmagatzemen amb aquest tipus d'elements. Els diferents estats, que poden ser l'estat del robot (posició, orientació, velocitat), l'estat del sensor (posició, orientació i intrínsecs) o dels landmarks (posició i orientació), no es tracten com a elements conjunts, sinó que es tracten en un vector com a elements separats o 'blocks'. Així en el moment de solucionar el problema i calcular els jacobians necessaris es pot treballar amb matrius *sparse* amb blocs i millorar l'eficiència computacional, a més de poder fixar blocs del vector d'estat i només estimar-ne els paràmetres que es desitgin. La dimensió d'aquest vector d'estats depèn del problema que s'estigui solucionant, ja sigui posició i orientació (PO 2D o 3D) o posició, orientació i velocitat (POV 3D).

Des d'aquesta classe es poden gestionar les Captures, afegint-ne més cada cop que se n'obtenen per part dels sensors o eliminant-ne si ja no són necessàries.

4.4.2 Capture

La classe **CaptureBase** és un objecte que té com a finalitat emmagatzemar les dades que arriben del sensor. És per aquest motiu que està molt lligat al sensor i té un *pointer* al sensor amb el qual s'ha obtingut. També conté un vector de state blocks amb posició, orientació i intrínsecs.

L'objecte Capture que s'obté amb el sensor és el que el processador fa servir per obtenir certes mesures interessants per la funció a desenvolupar i les emmagatzema en una llista de *features* que pengen de la captura a la qual pertanyen.

Particularització per al projecte

Per al projecte en concret, dels múltiples tipus de classes derivades que hi poden haver de CaptureBase (tantes com tipus de sensors ja que el tipus de dades van lligades al tipus de sensor, i per conseqüència també a la forma de processar la informació) interessa la classe CaptureImage.

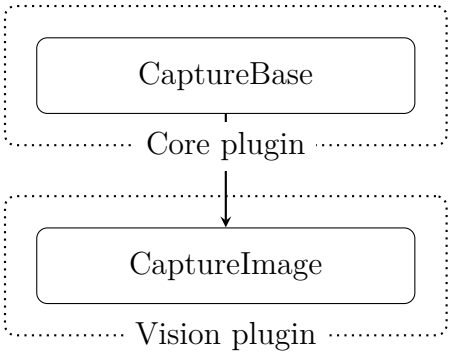

Figura 4.5: Classes derivades de Capture per al projecte

La classe CaptureImage conté tota la informació necessària a processar en aquest projecte:

- Punts característics de la imatge (KeyPoints) i descriptors de cada punt.
- Matches trobats entre punts d'aquesta imatge i la precedent i la seva qualitat.
- Matches a la imatge següent.
- Graella de features que divideix la imatge en diverses regions i es basa en la informació de les regions de la graella per saber si és necessari buscar-hi nous features o fer track amb una altra imatge i poder obtenir tracks repartits a tota la imatge.

4.4.3 Feature

La classe **FeatureBase** està pensada per contenir una mesura associada a la informació de la classe CaptureBase pare i es crea quan el processador processa les captures.

Per al projecte en particular, es farà servir una classe anomenada FeaturePointImage (veure Figura 4.6). Aquesta classe, bàsicament ha de contenir la informació:

– Mesura: serà un vector amb les coordenades del píxel mesurat.

– Identificador que associa el feature a un track que s'associa a un landmark.

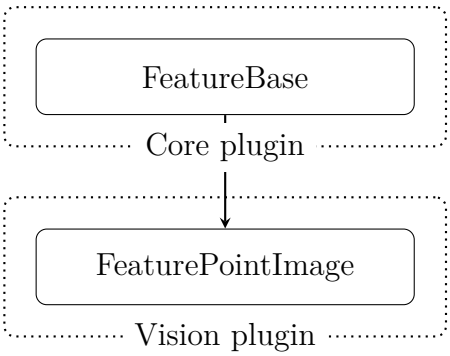

Figura 4.6: Classes derivades de Feature per al projecte

4.4.4 Factor

La classe **FactorBase** estableix un enllaç entre un Feature i algun altre element del mapa, que pot ser un altre Feature, un Landmark o un Frame. Per aquesta raó, aquestes tres classes anomenades tenen com a atribut una llista amb *pointers* als factors que hi tenen enllaçats.

Aquests enllaços els estableix el processador i és una classe essencial ja que estableix una relació directa entre la mesura i els state blocks que s'han d'estimar. A partir dels models d'observació dels sensors i els estats actuals, s'utilitzarà el factor per comparar la mesura amb el que s'esperava (esperança) i l'optimitzador intentarà minimitzar-ne l'error.

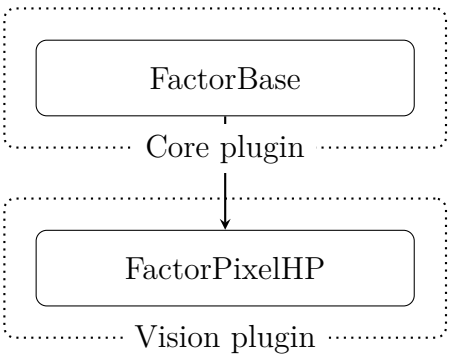

Figura 4.7: Classes derivades de Factor per al projecte

Per a aquest projecte en concret, s'ha creat la classe derivada FactorPixelHP (veure Figura 4.7). Aquesta classe implementa el tipus de restricció necessària per solucionar el

problema de bundle adjustment, que consisteix en minimitzar l'error de re-projecció entre el píxel mesurat i l'obtingut en projectar els landmarks al pla de la imatge. S'exposen els detalls al capítol 5.

4.5 Branca Map

La classe **MapBase** és la que conté la informació del mapa que configura l'entorn del robot a partir d'una llista que conté tots els landmarks observats a partir de les mesures dels sensors que té el robot. Cada cop que s'observa un nou landmark es pot afegir a la llista.

4.5.1 Landmark

La classe **LandmarkBase** és la que s'encarrega de donar forma a com són els landmarks observats. Derivant aquesta classe es poden crear molts tipus diferents de landmarks segons el problema a considerar, però com a elements bàsics tots ells comparteixen:

- Vector d'State Blocks amb la posició i orientació.
- Llista de restriccions amb els factors que associen el landmark amb un feature.

Per al projecte en concret, el tipus de landmarks que es faran servir són punts en l'espai, i s'ha implementat la classe LandmarkHP (veure Figura 4.8). Per a un punt en l'espai 3D es pot fer una parametrització de diverses maneres, i per aquest projecte s'ha fet d'una forma particular que es detalla en el capítol 5.

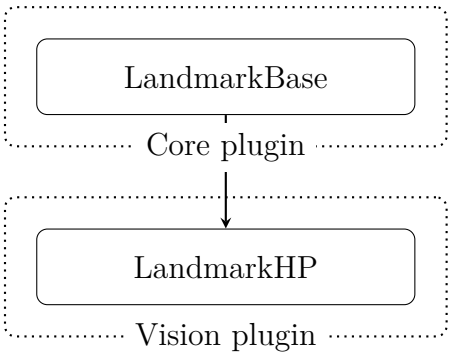

Figura 4.8: Classes derivades de Landmark per al projecte

4.6 Particularització per al projecte

Com s'ha explicat al llarg d'aquest capítol, Wolf és una llibreria per resoldre problemes de localització en robòtica, i un d'ells és l'SLAM. En el Capítol 2 s'ha explicat com es vol

resoldre el problema d'SLAM a partir d'aplicar un algoritme de bundle adjustment. Això és possible ja que en l'SLAM es busca estimar l'estructura del món exterior al mateix temps que la pròpia localització. En la formulació del problema de bundle adjustment, buscar la solució al problema equival a estimar l'estat òptim de localització pròpia i de l'entorn (que en aquest cas són landmarks corresponents a punts tridimensionals), per tant resoldre'l és equivalent a fer SLAM visual amb una càmera.

L'estructura de Wolf presenta una sèrie de classes base que permeten guardar tota la informació dels diferents elements que conformen el problema de l'SLAM. De fet, en la Figura 4.9 es pot observar com hi apareixen els mateixos elements que en el problema d'SLAM de la Figura 2.2 i que en el problema de bundle adjustment de la Figura 2.1. Si es para atenció a la connectivitat de l'arbre, es pot observar com gràcies als factors que connecten *features* amb els landmarks aquesta estructura és equivalent al factor graph de l'SLAM basat en landmarks que s'ha vist en la Figura 2.4.

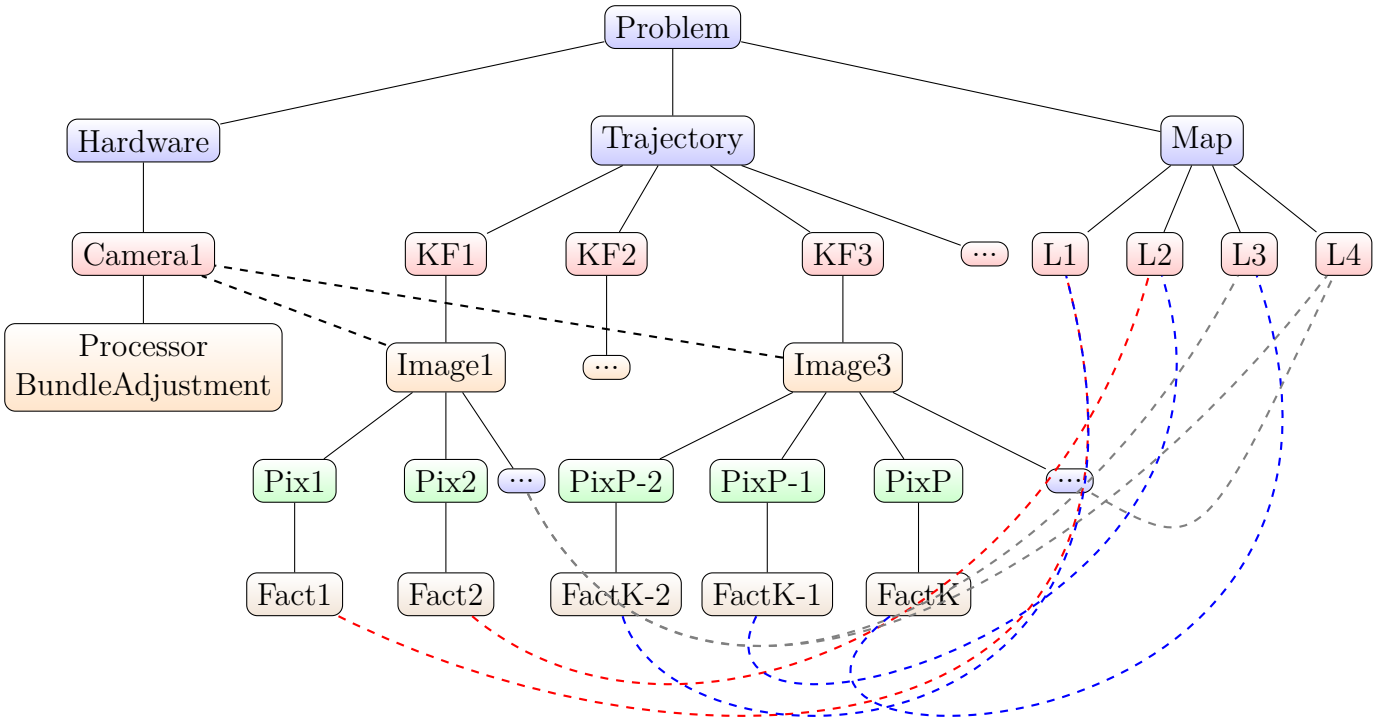

Figura 4.9: Arbre de Wolf amb els elements particulars del projecte.

De l'arbre de Wolf particular de la Figura 4.9 es poden veure totes les classes derivades que s'han d'implementar per al projecte. En el capítol 5, s'exposa com ha estat el desenvolupament d'aquestes classes. S'ha implementat la classe ProcessorBundleAdjustment, la classe FactorPixelHP i la classe LandmarkHP, i s'han utilitzat les classe de SensorCamera, CaptureImage i FeaturePointImage del plugin Vision per tal de tenir tots els elements del problema d'SLAM visual a resoldre pel *solver*.

Capítol 5

Contribucions a Wolf

En aquest capítol s'exposen les contribucions fetes a Wolf i es mostra com s'ha implementat el projecte per tal d'assolir els propòsits inicials. Les classes que s'han dissenyat són: ProcessorBundleAdjustment, LandmarkHP i FactorPixelHP.

5.1 Implementació del processador

El processador implementat a Wolf per tal d'assolir els objectius del projecte s'anomena ProcessorBundleAdjustment, i és una classe que deriva d'una herència de classes amb quatre capes d'abstracció, tal i com s'ha vist a la Figura 4.4 del Capítol 4.

Per tal d'entendre com s'ha implementat, en primer lloc és necessari veure les funcions que incorpora cada capa.

ProcessorBase

ProcessorBase és la classe base de qualsevol processador. Principalment té funcions genèriques que ha de tenir qualsevol tipus de processador per interactuar amb l'arbre de Wolf. Les funcions a destacar són dues **virtuals pures** que s'han d'implementar a les classes derivades:

- **process**: Aquesta funció rep com a argument un pointer a un objecte tipus captura, és a dir, una imatge en el cas d'aquest projecte, i ha de processar-la.
- **voteForKeyFrame**: Funció que retorna true si es valida el criteri per fer KeyFrame en l'última captura.

ProcessorTracker

Aquesta classe deriva de ProcessorBase i implementa un tracker incremental que pot ser utilitzat per fer track de Features en altres Captures o de Landmarks en el Map a través

de les classes derivades.

El **tracker incremental** té la intenció de fer un seguiment d'algun *feature* al llarg del temps comparant cada captura nova amb l'anterior. Tal i com està dissenyat a Wolf, conté tres pointers a tres captures de tipus base que s'anomenen *origin*, *last* i *incoming*:

- **Origin**: apunta a la Capture on el track de Features comença
- **Last**: apunta a l'última Capture on s'ha fet track per part del processador. Té Features d'*origin* que segueixen estant 'vius' en *last*.
- **Incoming**: apunta a la Capture que s'està rebent per part del sensor per ser processada. El processador estableix relacions entre els Features d'aquesta captura i els de les captures anteriors, ampliant la llargada dels tracks.

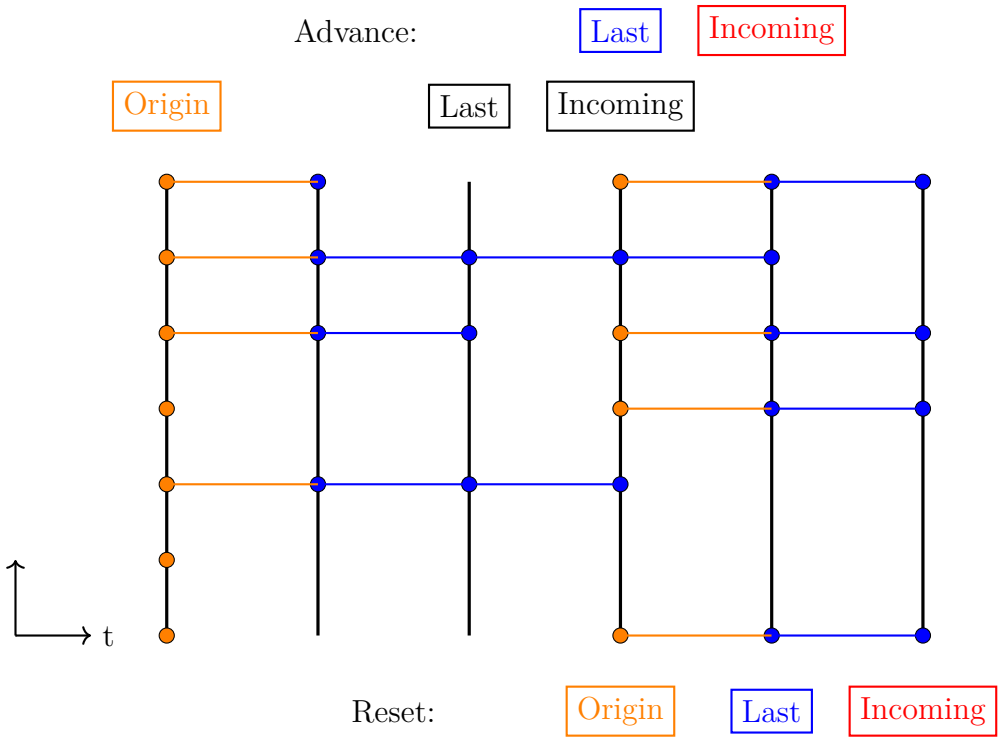

Figura 5.1: Tracker incremental

Aquesta classe implementa la funció **process()** que era virtual pura a ProcessorBase per fer el tracking incremental. Aquesta funció processa la Capture que arriba del sensor i segueix el procediment descrit a l'algoritme 2. Les funcions interessants a destacar són les que han de ser implementades per les classes derivades ja que són virtuals pures, i això està indicat amb un '= 0'.

Com a atributs a destacar, a part dels 3 *pointers* a les captures, té una llista amb els *new features* de *last* que s'han d'afegir al tracking incremental quan s'ha de fer KeyFrame i una llista amb els *new features* a *incoming* amb els quals s'ha establert un track provinent de *last*.

Degut a l'estructura del tracker d'utilitzar 3 captures, l'algoritme que segueix process() depèn de l'estat en el qual es trobi el tracker. En el cas específic del projecte, com no hi ha altres sensors a l'estructura de Wolf, el que pot passar és:

- **Primer cop**: es fa KeyFrame a la primera Captura, es processa i al final *origin* i *last* són al mateix lloc.
- **Secon cop**: es processa *incoming*, *last* passa a *origin* i *incoming* passa a *last*.
- **Running**: A partir de la tercera imatge ja es té cadascun dels 3 pointers a una captura diferent fins que es torni a fer KeyFrame.

Aquest procés de passar la captura *incoming* al pointer *last* es fa quan s'acaba de processar la captura *incoming* i s'anomena *advance* (veure Figura 5.1). Quan hi ha la necessitat d'establir KeyFrame, abans de fer aquest procés d'*advance* cal prèviament fer un procés anomenat *reset*, que consisteix en passar la captura de *last* a *origin* i continuar amb el tracking.

Algorithm 2: process() → A cada Incoming Capture es fa:

```
1  preProcess();
2  processKnown() = 0;
3  if voteForKeyFrame() then
4  |   processNew() = 0;
5  |   fer KeyFrame;
6  |   establishFactors() = 0;
7  |   reset() = 0;
8  |   advance() = 0;
9  end
10 postProcess();
```

ProcessorTrackerFeature

Aquesta classe deriva de la classe ProcessorTracker i implementa un tracker incremental que fa track de features entre les diferents captures. Aquesta classe implementa les virtuals pures que s'han vist en l'algoritme 2, que és el del mètode process().

La primera funció a considerar és **processKnown()**. Aquesta funció actua sobre la captura *incoming* i el que fa és fer track de features que estan vius entre aquesta captura i *origin*, però per fer-ho fa un track incremental a la captura *last* en comptes d'anar a buscar sempre l'origen del track. Això es pot veure a la Figura 5.1. Per tant, a la captura *last* sempre hi ha els tracks que han sobreviscut des d'*origin*. Aquesta funcionalitat l'implementa la funció trackFeatures().

En fer track cada cop amb la captura immediatament anterior (*last*), pot ser que els tracks vagin derivant cada cop una mica d'*origin* fins que al final no s'hi assemblin gens a causa de l'acumulació de petites diferències incrementalment, i per això hi ha una funció per corregir aquesta deriva anomenada correctFeatureDrift(). Aquests mètodes també són funcions virtuals a implementar en classes derivades.

Cal destacar que com a atributs aquesta classe té una llista de *known features* a *incoming*, que són els *features* als quals s'ha fet track des de *last*, i també té un objecte anomenat *TrackMatrix*. Aquest objecte consisteix en una matriu amb tota la història de tracks que permet un accés bidimensional, és a dir, accedir a tots els features d'un track entre dues captures o bé accedir a tots els features als quals s'ha fet track des d'una captura.

Algorithm 3: processKnown()

```
1 Buidar la llista de known features;
2 trackFeatures() = 0 ;
3 for cada track de trackFeatures trobat do
4 |   Afegir-lo a la track matrix;
5 end
6 Comprovar la deriva del track (correctFeatureDrift()=0);
7 Emplaçar els features a l'arbre de Wolf;
```

La següent funció a considerar és **processNew()**, i aquesta es crida només quan s'ha votat per KeyFrame, és a dir, quan en la captura *incoming* hi ha alguna condició com, per exemple, que hi ha pocs tracks vius respecte *origin*. A la Figura 5.1 es pot veure que quan hi ha pocs tracks a la captura que s'està processant, el que fa aquesta funció és fer KeyFrame a *last*, que passarà a ser *origin*, i es busquen nous tracks de *last* (que és on encara hi ha prou informació) cap a *incoming* per tornar a reomplir els tracks que s'havien perdut a partir de la funció detectNewFeatures(). Aquestes funcionalitats també s'han d'implementar en les classes derivades.

Algorithm 4: processNew()

```
1 Buidar la llista de new features;
2 detectNewFeatures() = 0 ;
3 for cada track de detectNewFeatures trobat do
4 |   Afegir-lo a la track matrix;
5 end
6 trackFeatures() = 0;
7 Emplaçar els features a l'arbre de Wolf;
```

Per últim, hi ha la funció **establishFactors()** que s'encarrega de crear un factor entre els features que s'han obtingut i algun altre element, que serà un landmark.

ProcessorBundleAdjustment

El processador Bundle Adjustment és la classe derivada de ProcessorTrackerFeature que s'ha implementat en aquest projecte i, per tant, una de les contribucions fetes a Wolf.

Aquest processador és el que gestiona com s'omple l'arbre de Wolf i les seves relacions per tal de fer SLAM visual amb un algoritme de bundle adjustment. Implementa les funcions virtuals a sobre de les classes abstractes sobre les que està constituït.

L'objectiu d'aquest processador és fer un tracking incremental de features presents en cada imatge que arriba de la càmera amb la imatge immediatament anterior, i per fer això implementa el disseny de les seves classe pare. Al mateix temps, també es pretén crear una relació entre cada feature i el landmark present a l'exterior, ja que el problema de bundle adjustment necessita minimitzar l'error de re-projecció entre el feature detectat i el landmark re-projectat, de manera que s'estableixen factors que 'lliguen' cada track de features amb un landmark. Així s'estan aconseguint dues coses: fer track de features a través d'imatges i, a través d'això, fer un track de landmarks a l'espai que permetrà resoldre el problema de bundle adjustment i, per tant, resoldre el problema d'SLAM a través de Wolf.

En les següents seccions s'exposa la implementació de cada una de les funcions de la capa més externa del processador.

Constructor

El constructor de la classe ProcessorBundleAdjustment bàsicament està dedicat a inicialitzar el processador amb els paràmetres que li corresponen. A més, és en el punt on s'inicialitza la tríada detector-descriptor-match que s'ha escollit al Capítol 3 i, per tant, és el punt de connexió entre la feina feta en la primera part del projecte i l'algoritme que s'ha desenvolupat en aquest capítol.

voteForKeyFrame

La funció **voteForKeyFrame()** és l'encarregada d'avaluar si la condició escollida per fer KeyFame es compleix, i en cas que sí el processador ho farà. Un KeyFrame en l'estructura de Wolf és una pose del robot clau en la reconstrucció de la seva trajectòria, i es pot escollir fer-ne seguint criteris variats.

En aquest projecte, com que la informació prové del track de features en les imatges, s'ha establert que es faci un KeyFrame cada cop que es passa per sota un llindar de tracks mínim, per evitar perdre la història de la trajectòria del robot (i de la càmera que porta).

preProcess i postProcess

El mètode **preProcess()** implementa tota la funcionalitat d'OpenCV de manera que l'algoritme de Wolf només ha de gestionar la informació obtinguda i no ha de fer visió per computador.

Algorithm 5: preProcess()

```
1 Obtenir Capture Incoming;
2 Detectar KeyPoints OpenCV;
3 Calcular Descriptors OpenCV;
4 Inicialitzar Feature Grid;
5 if Last != nullptr then
6 |   Trobar Matches Last ← Incoming ;
7 |   Filtrar ambigüitats
8 end
```

A cada imatge que arriba, fent servir el mètode de detect-descript-match seleccionat, detecta tots els features d'OpenCV (KeyPoints), calcula els descriptors i fa match amb la imatge a *last* si és que n'hi ha alguna. Tota aquesta informació queda emmagatzemada dins les captures *incoming* i *last*. Quan es fa el match, *incoming* ha de tenir emmagatzemats els matches de la captura anterior i *last* ha de tenir un mapa que assigni a cada feature de *last* un feature de *incoming* si s'ha trobat.

Un problema que s'ha detectat en el moment de fer match utilitzant *FLANNBASED* és que es compara un feature de *incoming* contra tots els features de *last* i es selecciona el millor. Això pot produir que dos features de *incoming* vagin a parar al mateix *feature* de last i, mirant una seqüència d'unes quantes imatges, el que es veuria es com els tracks es van ramificant. Per evitar aquestes ambigüitats, el que s'ha fet és filtrar-les un cop fet el match d'OpenCV per tal que la resta de l'algoritme no tingui aquest conflicte.

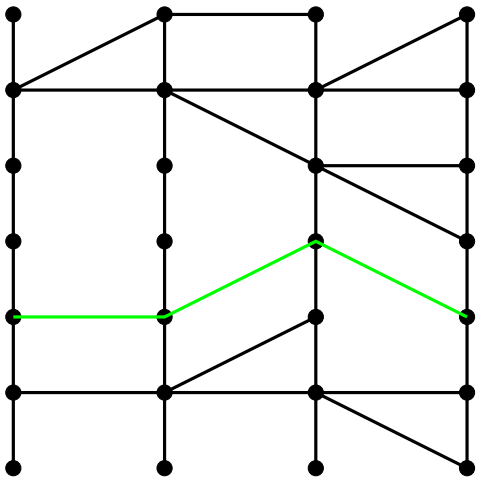

Figura 5.2: Ramificació dels tracks. En verd, track sense ramificar. En negre, tracks ramificats en fer match: last ← incoming.

Un cop es té la part d'OpenCV feta, l'algoritme crea una graella anomenada **Feature Grid** que està implementada a vision_utils i que està associada a cada imatge. La funcionalitat d'aquesta graella (veure Figura 5.3) és dividir la imatge en diferents regions amb tots els features d'OpenCV detectats. D'aquesta manera, quan s'ompli la graella amb features de Wolf, es farà de tal manera que quedin repartits per tota la imatge, i al mateix temps es podrà evitar buscar tracks o nous features en graelles on ja se sap que no es trobarà res del que es busca. Bàsicament el seu funcionament es basa en dos processos:

- **Hit**: es fa sobre una cel·la de la graella a *last* s'ha trobat un track cap a *incoming*. Així s'evita tornar a buscar en aquesta zona on ja hi ha un track.
- **Block**: es fa quan sobre una cel·la no s'ha trobat cap track de Wolf per tots els features d'OpenCV, de manera que no cal buscar-hi més.

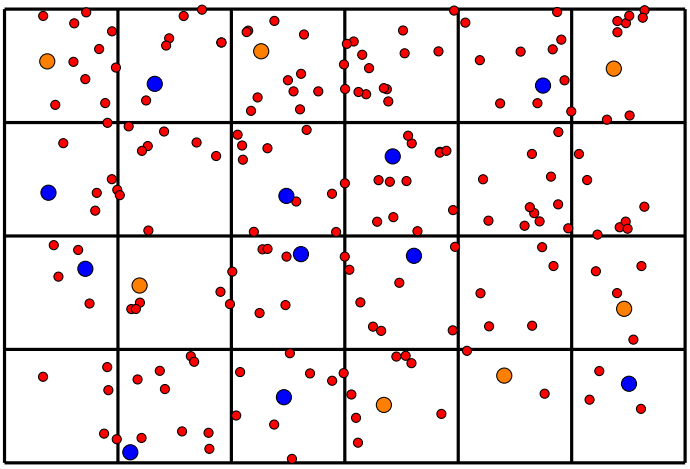

Figura 5.3: Feature Grid. En vermell es mostren els features d'OpenCV que té la imatge. En blau són els features de Wolf sobre els quals es té un track establert (aquestes caselles estaran en *hit*). En taronja es mostren els features pels quals existeix track amb *incoming*; aquestes caselles són on l'algoritme pot buscar *new features* i quan en troba fa *hit* per no tornar-hi a buscar, però de moment es poden utilitzar. En les caselles on no hi ha ni punt blau ni taronja és perquè no cal buscar-hi, s'haurà de fer un *block* ja que no és possible establir un track amb features de Wolf.

El mètode **postProcess()** fa principalment la funció de visualitzador 2D del track de features al llarg de la seqüència d'imatges. Per fer-ho agafa tots els tracks de la imatge a visualitzar a partir de la *track matrix* i dibuixa una línia corresponent a cada feature del track que dona una idea del moviment de la càmera. També es dibuixa la re-projecció dels landmarks. Per fer la visualització es fa ús de les funcions implementades a vision_utils.

A la Figura 5.4 es pot observar la visualització 2D. Les línies blaves representen els tracks que s'han establert al llarg d'unes quantes imatges, el punt vermell és la re-projecció del landmark i en els llocs on hi ha la creu i un número (que és l'identificador i la llargada del track) és on es troba el feature de Wolf actual.

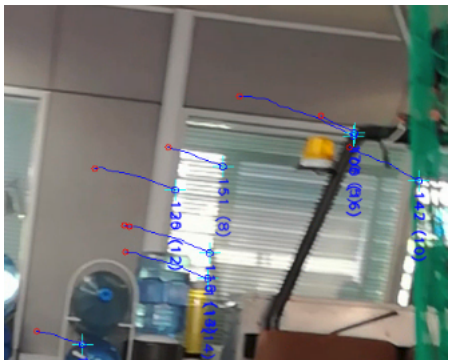

Figura 5.4: Visualització postProcess()

trackFeatures

Aquesta funció s'encarrega de fer track de features de *last* cap a *incoming*. Per fer-ho, s'aprofita de la feina que s'ha fet d'OpenCV al preProcess() pe no haver de tornar a detectar ni a fer match. El que fa és que per als features de Wolf existents a *last*, es mira si aquests tenien un match d'OpenCV amb *incoming* establert durant el preProcess, i si és així crea un feature de Wolf a *incoming* i d'aquesta manera s'allarga el track.

Algorithm 6: trackFeatures() → Per cada feature a last omple una llista de features a incoming si es manté el track

```
for cada Feature de Last do
    if existeix track del feature a Incoming then
        Obtenir índex i keypoint OpenCV del Feature Incoming;
        Crear un FeaturePointImage de Wolf;
        Omplir mapa de correspondències feature - match;
        Avisar a Feature Grid que hi ha un tracked point de Wolf (hit)
    end
end
```

detectNewFeatures

Aquesta funció serveix quan s'ha de fer KeyFrame a *last* perquè s'ha votat per KeyFrame. Aleshores s'han de buscar nous features a *last*, que és on encara es té prou informació, que segueixin vius a *incoming*. Per fer-ho, es fa ús de la Feature Grid, ja que per una banda en les cel·les on ja hi ha features de Wolf no es busquen nous tracks per tenir-ne de repartits per tota la imatge (es diu que s'havia fet *hit* a la cel·la). Per altra banda, només busca tracks a les cel·les on només hi ha features d'OpenCV i que a més tenen match d'OpenCV contra *incoming* trobat durant postProcess, ja que en cas contrari la cel·la estaria bloquejada (*block*).

Algorithm 7: detectNewFeatures() → té com a input un nombre màxim de features

```
for iteració de i=0 fins a max number of features do
    if existeix cel·la buida a last Feature Grid then
        Obtenir una cel·la buida;
        Obtenir els millors features de la cel·la;
        for cada feature OpenCV de la cel·la do
            if hi ha match del feature a Incoming then
                Crear un feature de Wolf a last;
                Avisar a Feature Grid que hi ha un tracked point de Wolf (hit);
                break;
            end
        end
        if no s'ha trobat feature la cel·la then
            Avisar que a la cel·la no hi poden haver features per fer track (block)
        end
    else
        break;
    end
end
```

Creació de factors i de landmarks

El mètode del processador que permet establir una restricció entre els features de cada track i els landmarks del món exterior s'anomena *establishFactors()*, que és una funció sobreescrita provinent d'una altra capa del processador. Per tal de poder crear els factors, s'ha tingut en compte que només pot existir un landmark per cada track de features i que per cada feature del track ha d'existir un factor cap al landmark. Per aquest motiu, el processador té com a atribut un mapa de landmarks on accedint amb l'identificador del track s'accedeix al landmark al qual correspon. El procediment seguit està descrit a l'algoritme 8.

Com es veu en l'algoritme 8, es crida a la funció *createLandmark()* cada cop que un track és nou. Aquesta funció està implementada en aquesta capa del processador i rep com a argument un Feature de la imatge. A partir d'aquest feature i el model de projecció de la càmera, projecta el píxel a l'espai tridimensional transformant-lo a la referència adequada i crea el landmark.

Algorithm 8: establishFactors()

```
1  Obtenir tracks entre Origin i Last de la track matrix;
2  for cada track entre Origin i Last do
3      Obtenir track id i Features;
4      if el track té un landmark associat then
5          createLandmark();
6          Crear Factor Landmark - Feature Origin;
7          Crear Factor Landmark - Feature Last;
8          Emplaçar Factors a l'arbre;
9      else
10         Obtenir landmark associat al track;
11         Crear Factor Landmark - Feature Last;
12         Emplaçar Factor a l'arbre;
13     end
14 end
```

5.2 Implementació dels landmarks

El tipus de landmark que apareix en el projecte és un punt tridimensional a l'espai que es tradueix en un píxel a la imatge. En aquesta transformació d'un punt de l'espai 2D de la imatge a l'espai 3D, hi ha un grau de llibertat que no és observable i es perd la noció de distància. Per representar-ho, aquest landmark pot prendre diverses formes i per al projecte s'ha decidit representar els landmarks com a *homogeneous points*. Per implementar-ho s'ha utilitzat la parametrització que es es basa en [25].

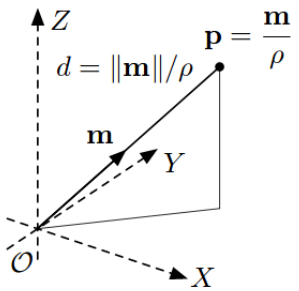

Figura 5.5: Homogeneous Point (HP). Font: [25]

Un punt homogeni (*homogeneous point HP*) es parametritza com un vector de 4 components en l'espai projectiu tridimensional, que està format per un vector $\mathbf{m}$ de 3 components i un escalar ρ.

$$\mathcal{L}_{HP} = \underline{\mathbf{p}} = \begin{bmatrix} m_x & m_y & m_z & \rho \end{bmatrix}^T \tag{5.1}$$

El punt en coordenades cartesianes, aleshores, té l'expressió:

$$\mathbf{p} = \frac{m}{\rho} \tag{5.2}$$

A partir d'aquest base es poden escollir diferents alternatives. Treballar amb $\rho = 1$ equival a la representació original del punt en coordenades cartesianes. Una altra alternativa és treballar amb l'opció de $\|m\| = 1$, on aleshores el paràmetre escalar ρ passa a representar l'invers de la distància. D'aquesta manera $\mathbf{m}$ és un versor en una direcció de l'espai, i dividint-lo per l'invers de la distància ρ s'obté el punt en coordenades cartesianes. Amb aquesta parametrització, el grau de llibertat no observable, que és la distància, queda recollit en el paràmetre ρ, que també s'anomena la part homogènia del punt.

Tal i com està fet a Wolf, s'ha escollit treballar amb $\left\|\underline{\mathbf{p}}\right\| = 1$, de manera que els landmarks són punts homogenis 3D de norma fixa. Aleshores ρ és proporcional a l'invers de la distància. D'aquesta manera es defineix una esfera 4D en l'espai projectiu i té el benefici que es pot utilitzar l'àlgebra del quaternió per treballar-hi quan s'optimitza el seu estat.

Un cop la parametrització escollida, per projectar un punt de l'espai a la imatge o fer el procés contrari s'han de fer servir els models matemàtics de projecció que segueix el model de la càmera. Aquestes eines, a Wolf, es troben al que s'anomena *pinholetools*, un conjunt de funcions que permeten fer la part matemàtica relacionada amb els model de càmera pinhole i que té funcions per projectar (3D a 2D), back-project (2D a 3D), obtenir els paràmetres de distorsió de la càmera i corregir-la, i *pixellize* les coordenades dels punts bidimensionals.

A Wolf, el landmark s'ha implementat a la classe LandmarkHP, que deriva de la classe LandmarkBase. El constructor d'aquesta classe rep com a paràmetre un vector de 4 components corresponent a la parametrització del landmark com a punt homogeni, i ho emmagatzema a wolf com a tipus *StateHomogeneous3D*. Aquest tipus com a classe base és un State Block que emmagatzema la posició del landmark amb quatre components.

En l'API de la classe, hi ha implementada un funció anomenada *point()* que fa l'operació descrita a l'equació 5.2 per obtenir les coordenades cartesianes.

Un punt important de la implementació dels landmarks és el moment de creació, que està implementat al processador perquè és qui s'encarrega de fer-ho. En aquell moment és quan es fa la composició de referències per representar el landmark en coordenades globals. El procés que es segueix en el mètode *createLandmark()* es descriu a l'algoritme 9.

Algorithm 9: createLandmark() → Input: Feature que correspon a un punt de la imatge

1 Obtenir la mesura del feature corresponent al punt 2D de la imatge;
2 Calcular el punt3D a l'espai fent back project del punt2D amb pinholetools;
3 Obtenir el punt homogeni a partir $\|punt3D\|$ i un paràmetre arbitrari de distància;
4 Transformar el landmark de referència càmera C a referència món W passant pel robot R : ${}^{W}\underline{\mathbf{p}} = T_{WR}\, T_{RC}\, {}^{C}\underline{\mathbf{p}}$;
5 Construir el LandmarkHP;
6 Emplaçar el LandmarkHP a l'arbre de Wolf;

5.3 Implementació dels factors

L'últim element que falta per implementar al 'puzzle', i un dels més importants, és el *factor*. Aquest element imposa la restricció entre el landmark i el feature a la imatge que permetrà minimitzar l'error de re-projecció i, per tant, en aquest element és on està implementat el bundle adjustment.

El factor que s'ha implementat és una classe anomenada FactorPixelHP. El nom prové dels dos elements que enllaça, un píxel de la imatge amb un punt homogeni en l'espai. Així doncs, al constructor d'aquesta classe cal passar-l'hi bàsicament els *pointers* al landmark i el feature als quals enllaça, i a partir d'aquí crea la restricció.

La classe FactorPixelHP ha d'implementar l'error de re-projecció que es vol minimitzar en el bundle adjustment (secció 2.1) i que s'ha exposat en l'equació 2.3.

Els dos elements d'aquesta equació són:

- **Mesura**: correspon a les coordenades 2D mesurades a la imatge.
- ***Expectation***: correspon a l'esperança del valor de la mesura. S'obté projectant el punt homogeni del landmark sobre el pla de la imatge i obtenint les coordenades 2D.

La mesura s'obté directament del Feature, que la té emmagatzemada, però per obtenir l'*expectation* hi intervenen més elements. L'error és la resta de la mesura i l'*expectation*, i la mesura està fixada pel píxel vist a la imatge, així que l'element clau del factor és l'*expectation*, que depèn del model d'observació de la càmera, és a dir, que és la projecció d'un punt de l'espai sobre la imatge. En aquest model és on precisament hi intervenen els elements que es volen estimar en l'SLAM: la localització del robot(i de la càmera sobre el robot) i la localització del landmark.

Per calcular-ho, la classe FactorPixelHP té una funció anomenada **expectation()** que rep com a arguments els state blocks que s'han d'estimar, ja que depenent de com canviïn també canviarà l'*expectation* a comparar amb la mesura. Els inputs són:

– Posició del robot al món: t_{WR}

– Orientació del robot al món: q_{WR}

– Posició del sensor al robot: t_{RC}

– Orientació del sensor al robot: q_{RC}

– Posició del landmark com a punt homogeni al món (4 components)

Aquests inputs són els que el *solver* s'encarrega de modificar en el *back-end*, i a cada iteració els modifica i recalcula l'*expectation* perquè s'assembli el màxim possible a la mesura.

Per calcular l'*expectation* a partir dels inputs cal fer una composició de referències per tenir el landmark expressat en coordenades de la càmera i utilitzar les pinholetools per projectar-lo sobre la imatge.

$$ {}^{C}\underline{\mathbf{p}} = T_{CR}\, T_{RW}\, {}^{W}\underline{\mathbf{p}} \tag{5.3} $$

on la transformació T_{CR} s'obté fent la inversa de la composició de t_{RC} i q_{WR}, i T_{RW} s'obté de la inversa de la composició de t_{WR} i q_{WR}.

Un cop es té l'*expectation* i la mesura, la classe té una sobrecàrrega de l'operador **( )** per obtenir el **residu** fent la resta: **mesura - expectation**.

Per últim, cal comentar que la classe FactorPixelHP deriva de la classe implementada a Wolf que s'anomena FactorAutodiff. Aquesta és una classe templatitzada que s'encarrega de treballar amb els jacobians dels state blocks a estimar de manera automàtica.

Capítol 6

Resultats

En aquest capítol s'exposen els resultats obtinguts i el funcionament dels algoritmes del sistema d'SLAM visual.

6.1 Tríada detector-descriptor-matcher

Els resultats d'aquest apartat ja s'han discutit en la secció 3.6.4 del Capítol 3, on s'han mostrat els gràfics amb els resultats de les mètriques que s'han fet servir per seleccionar la tríada adequada. Aquesta tríada ha estat configurada amb els paràmetres que es veuen a la Taula 6.1.

Paràmetre	Valor escollit
nfeatures	400
scale factor	1.0
nlevels	1.0
edge threshold	31
WTA_K	2
score type	1
patch size	31

Taula 6.1: Paràmetres detector-descriptor ORB

A la Taula 6.2 es poden visualitzar els valors finals de les mètriques que s'han considerat per seleccionar l'alternativa final. Cal comentar que aquests resultats han estat obtinguts amb un ordinador amb un processador de vuit nuclis i 8GB de memòria, pel que poden variar segons la capacitat de còmput de cada màquina.

Especificació	Mètrica	Valors
Rapidesa	temps t (ms)	36,6 ms
Longitud dels tracks	Número de *features* per track N	N>2
Invariabilitat a l'escala	Número d'octaves O	0

Taula 6.2: Taula d'especificacions de la tríada

Un cop s'ha tingut la tríada configurada, s'ha provat d'utilitzar-la en imatges en blanc i negre i s'ha obtingut una velocitat d'execució en el temps complet de detecció-descripció-match que no supera els 15 ms, de manera que s'ha comprovat com s'estalvia temps de còmput i es pot utilitzar per aplicacions en temps real. A la Figura 6.1 es pot veure el funcionament de la tríada.

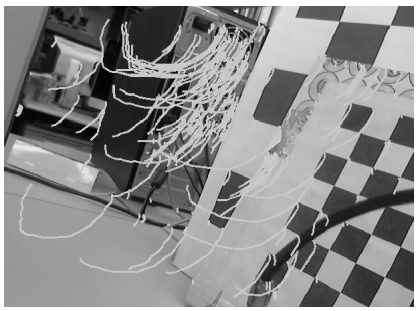

Figura 6.1: Tracks obtinguts amb els paràmetres ajustats fent un moviment circular amb la càmera.

6.2 Tests unitaris de l'algoritme a Wolf

Per tal d'avaluar els resultats i funcionament de l'algoritme implementat a Wolf, s'han realitzat tests unitaris de cada apart de l'algoritme per tal d'analitzar-ne el funcionament. Aquests tests són en la gran majoria comprovacions senzilles per assegurar que cada element de l'algoritme fa el que li correspon, però com a tests més importants i que involucren diverses parts es poden destacar la visualització 2D i el test al factor.

Test al factor

El test més important a destacar és el que s'ha realitzat sobre la classe FactorPixelHP. S'ha simulat un *setup* amb tres frames corresponents a tres poses de la càmera que observen a 4 landmarks.

Amb el *setup* preparat, s'ha construït un problema de Wolf PO 3D (per estimar posició i orientació) amb el processador ProcessorBundleAdjustment i un SensorCamera instal·lats i s'han fixat la pose d'una càmera i un landmark per tal que els graus de llibertat siguin observables.

S'ha fet un solve del problema pertorbant amb un valor aleatori la posició dels frames i el landmark no fixats i fent servir com a opcions del solver un màxim de 200 iteracions i una tolerància de 1e-6.

El resultat ha estat que el solver convergeix a la solució correcta en els termes que es visualitzen a la Figura 6.2. Això demostra que l'algoritme de bundle adjustment funciona completament i el solver és capaç de convergir a la solució correcta.

De la Figura 6.2 es pot destacar que el solver ha convergit en 20 iteracions passant d'un cost inicial molt elevat a un cost pràcticament nul.

```
Solver Summary

                                     Original                  Reduced
Parameter blocks                           13                        7
Parameters                                 48                       26
Effective parameters                       40                       21
Residual blocks                            12                       11
Residuals                                  24                       22

Minimizer                        TRUST_REGION

Sparse linear algebra library    SUITE_SPARSE
Trust region strategy     LEVENBERG_MARQUARDT

                                       Given                     Used
Linear solver          SPARSE_NORMAL_CHOLESKY    SPARSE_NORMAL_CHOLESKY
Threads                                    1                        1
Linear solver ordering                AUTOMATIC                     7

Cost:
Initial                         1.619624e+06
Final                           1.279107e-12
Change                          1.619624e+06

Minimizer iterations                      20
Successful steps                          15
Unsuccessful steps                         5

Time (in seconds):
Preprocessor                    0.000024

  Residual only evaluation      0.000095 (20)
  Jacobian & residual evaluation 0.001083 (15)
  Linear solver                 0.000221 (20)
Minimizer                       0.001551

Postprocessor                   0.000003
Total                           0.001578

Termination:                   CONVERGENCE (Parameter tolerance reached. Relative step_norm:
    1.899962e-09 <= 1.000000e-08.)
```

Figura 6.2: Solver Summary obtingut del test de l'algoritme de bundle adjustment.

Visualització 2D

Un dels altres tests realitzats ha estat una demo per tenir una visualització 2D del problema, on es veuen els tracks i la projecció dels landmarks sobre la imatge utilitzant l'algoritme de Wolf (veure Figura 6.3).

En aquesta demo s'ha pogut comprovar de manera visual com es van mantenint els tracks al llarg de diferents imatges i com cada cop que es perden tracks per sota un nombre

establert en els paràmetres del processador es fa un keyframe i es busquen nous tracks. També s'ha pogut veure com a cada track hi ha una re-projecció d'un landmark associat, i que en fer solve del problema la distància entre la re-projecció en la imatge i el píxel més actual del track es redueix.

Per tal de tenir una bona visualització, amb les eines de vision_utils s'han pintat tots els punts de cada track que encara és viu en l'última imatge per donar una idea del moviment de la càmera. A l'extrem de cada track hi ha una creu amb el píxel més actual del track que té al costat un número corresponent a l'identificador i un altre corresponent a la longitud. Cada track té una re-projecció d'un landmark en vermell si té longitud prou llarga com perquè s'hagi creat el landmark, i a cada landamrk se l'hi pot dibuixar el seu identificador.

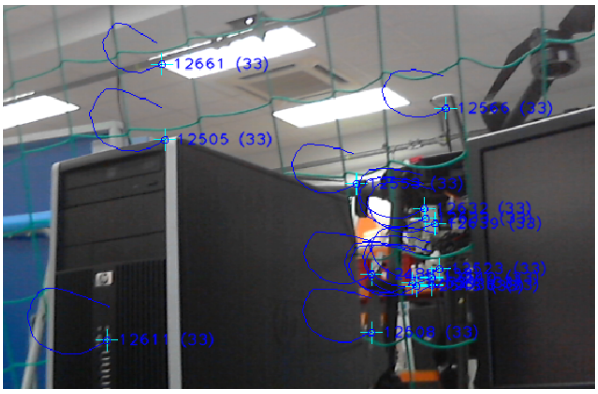

Figura 6.3: Visualització 2D a Wolf fent moviments circulars amb una càmera. En blau, els tracks obtinguts. El número al costat de la creu que hi ha a cada track és l'identificador del track, i el número entre parèntesi el número de frames que dura.

6.3 Avaluació en un node de ROS

Una vegada comprovat que la tríada detector-descriptor-matcher escollida permet processar les imatges a temps real i comprovat amb els resultats dels tests unitaris a Wolf i la visualització 2D que l'algoritme dissenyat funciona tal i com s'espera, s'ha procedit a realitzar una comprovació global del projecte a partir d'un problema real. Per fer-ho, s'ha implementat un node de ROS per tal de poder visualitzar en 3D el problema.

En primer lloc, s'ha utilitzat una càmera industrial *global shutter* i se n'han obtingut els paràmetres de calibració, s'han ajustat els paràmetres de visió de la càmera (com temps d'exposició o enfocament entre d'altres) per adaptar-la a les condicions de llum de l'escena del laboratori de robòtica mòbil de l'IRI i s'ha instal·lat el driver necessari i un node de ROS que permet obtenir les imatges que es publiquen provinents de la càmera.

Amb la part de la càmera configurada, s'ha implementat un node que basat en la feina feta a l'IRI té una estructura que implementa per una banda el node de ROS en si mateix

i per l'altra un *wrapper* de l'algoritme de Wolf per obtenir la informació per configurar i visualitzar el problema així com la informació per solucionar-lo.

Visualització 3D

La visualització 3D en el node de ROS s'ha fet a partir del visualitzador 3D anomenat RViz, que s'ha configurat de tal manera que mostri la informació que es publica des del node de ROS.

S'ha configurat el node per tal de publicar:

- Esferes 3D corresponents a la posició dels landmarks amb el número identificador a sobre.
- Cubs corresponents a les posicions de la càmera.
- Línies Cub-Esfera que representen els factors entre una feature associat a una pose de la càmera i un landmark.

A partir d'això, s'ha gravat un rosbag amb un moviment de la càmera i s'ha fet servir per tal de processar-lo pas per pas i analitzar la convergència de la solució.

Per una banda, observant pas per pas la visualització 3D obtinguda i comparant-ho amb la visualització 2D a Wolf, s'ha vist com quan es creen landmarks aquests es visualitzen en 2D amb un identificador i que al mateix temps es situen a l'espai 3D a RViz en una configuració que es correspon al que s'observa a la imatge, amb un factor per cadascun d'ells. D'aquesta manera, es pot comprovar com l'algoritme és capaç de fer track incremental, associar els tracks a landmarks observats i crear els factors corresponents, de manera que el processador omple l'arbre de Wolf i es configura correctament el problema en un entorn real.

Comparant les Figures 6.5 i 6.4 es pot veure com els landmarks associats a cada track que estan projectats en la imatge 2D es corresponen als landmarks a l'espai en la visualització 3D.

Per altra banda, pel que fa referència a la solució del problema que ja s'ha vist que es configura correctament, s'ha observat que en passar la informació per l'optimitzador no es convergeix a una solució que reprodueixi la realitat del problema. El resultat obtingut solucinant el problema és lògic ja que en el moment de redacció de la memòria s'està treballant en millorar certes parts del 'puzzle' global del problema. S'estan realitzant les tasques de millorar el calibratge de la càmera i la feina de calcular i proporcionar un *prior* de la *pose* del frame per tal que en iterar es parteixi d'una situació propera a l'òptim.

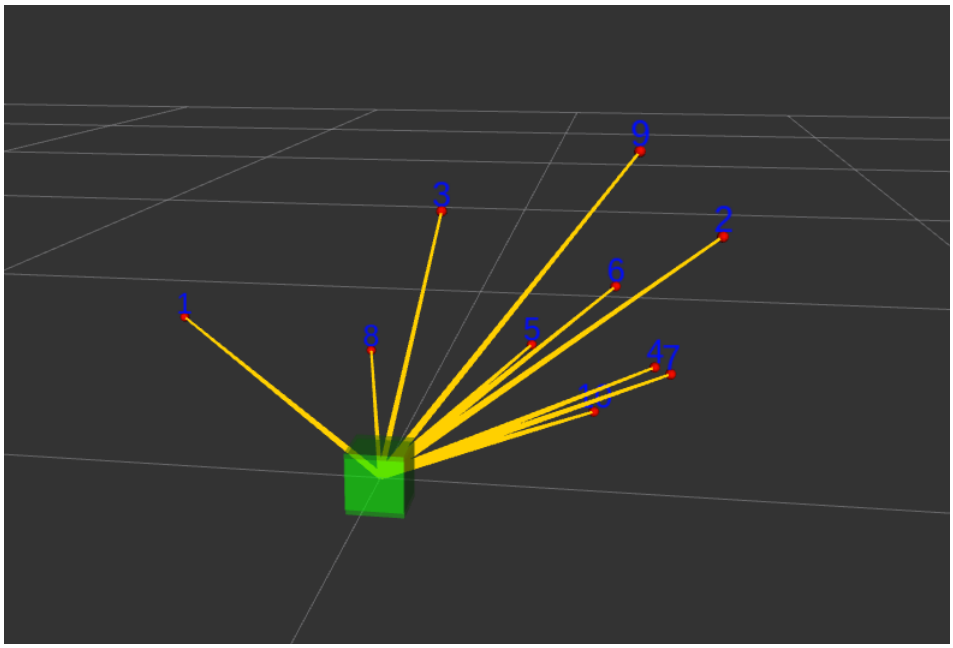

Figura 6.4: Visualització 3D a RViz. Els cubs verds representen les poses de la càmera, els punts vermells els landmarks amb l'identificador en blau, i les línies grogues els factors establerts.

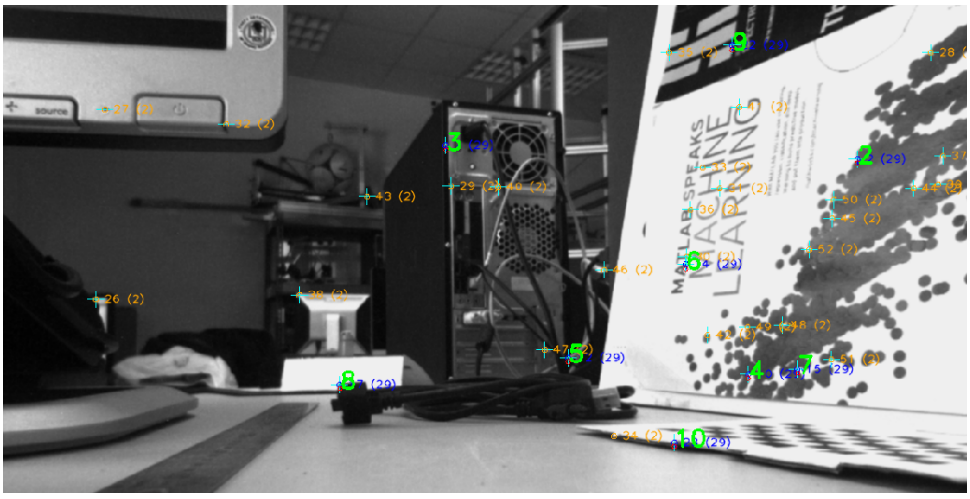

Figura 6.5: Visualització 2D. En taronja es dibuixen els nous tracks obtinguts en fer keyframe. En blau els tracks que ja es tenien. Cada track té un identificador i la llargada en imatges entre parèntesi. En vermell la re-projecció dels landmarks sobre la imatge. En verd es veu el número que identifica cada landmark. En aquest frame el landmark número 1 s'ha perdut i no apareix.

Estudi econòmic i Planificació

En aquest apartat s'exposa la planificació que s'ha seguit per realitzar el projecte i es fa un recull de les despeses econòmiques derivades de la seva elaboració. El projecte ha estat completat en un període de 18 setmanes i a la Figura 6.6 es pot veure un diagrama de Gantt amb les fases del projecte.

Planificació

Per a la planificació del projecte, s'ha fet una divisió de les tasques en cinc grups de treball:

- **Estudi previ de les eines necessàries**

  En aquesta fase s'ha fet un estudi de les eines necessàries per desenvolupar el projecte. Dins d'aquestes tasques si engloba aprendre a programar en C++, *ROS*, sistemes de control de versions (*git*) i l'estudi d'OpenCV i de tècniques de visió per computador, a més d'altres eines bàsiques.

- **Tríada detector-descriptor-matcher**

  En aquesta fase s'hi troben les tasques relacionades amb fer un estudi de totes les opcions de detectors, descriptors i matchers, fer un estudi de vison_utils, desenvolupar un algoritme per fer *tracks* en seqüències de vídeo i fer una anàlisi de les dades obtingudes per seleccionar la millor combinació.

- **Wolf**

  En aquesta fase s'han fet, per una banda, tasques teòriques d'estudi del codi de la llibreria Wolf i l'estudi de la formulació matemàtica del problema de Graph SLAM i de bundle adjustment. Per altra banda, s'han fet tasques pràctiques de desenvolupar l'algoritme per fer SLAM visual dins la llibreria Wolf i testos per comprovar la bondat del codi desenvolupat.

- **Anàlisi dels resultats**

  En aquesta fase s'hi comprenen les tasques relacionades en crear un node de *ROS* per fer SLAM visual, calibració de la càmera i ús dels drivers necessaris i creació de visualitzadors 2D i 3D per analitzar el funcionament de la solució creada.

- **Documentació**

  En aquesta fase del projecte s'hi engloba tota la feina relacionada amb la redacció de la memòria.

Per fer una planificació temporal d'aquestes tasques, s'ha fet un diagrama de Gantt on es mostra el temps dedicat a cada fase del projecte entre els mesos de Febrer a Juny, al llarg de 18 setmanes en total.

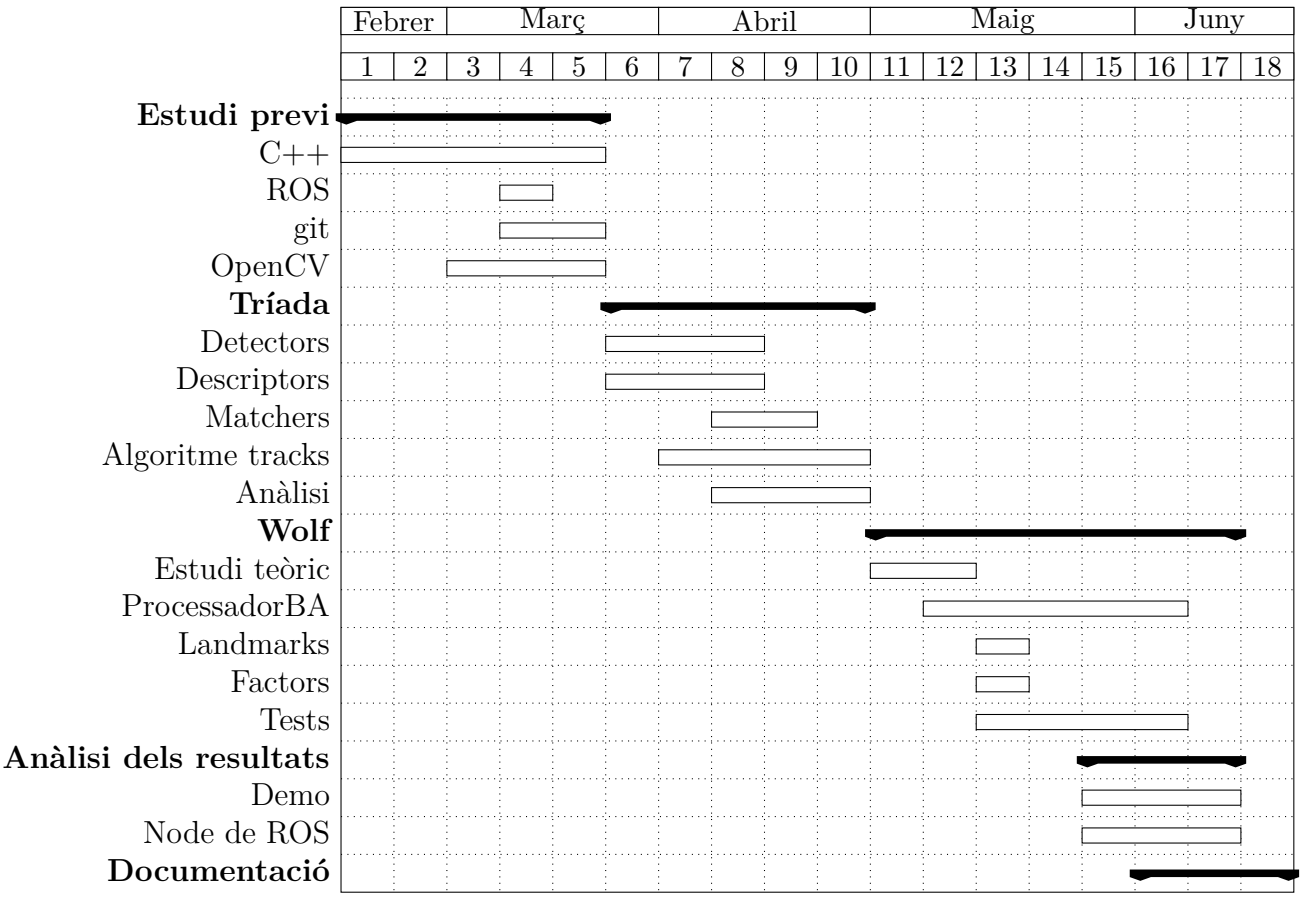

Figura 6.6: Diagrama de Gantt del projecte

Per tal de calcular el pressupost del projecte s'han tingut en compte dos tipus de costs segons els recursos utilitzats: per una banda recursos humans i per l'altra els recursos materials.

Costs de recursos humans

Pel que fa als recursos humans, s'ha considerat que s'han fet diferents tipus de feina i que, per tant, segons el tipus de feina el cost horari és diferent. Aquesta classificació es pot veure a la Taula 6.3. En els quatre tipus de feina que s'han tingut en compte, el tipus 1 correspon a la feina d'enginyeria de disseny i inclou el disseny d'algoritmes i

l'anàlisi crítica dels resultats. El tipus 2 correspon a feina de programador i inclou la implementació del codi a Wolf i el codi en la part de la selecció de la tríada. El tipus 3 correspon a feina administrativa de redacció de la memòria i el tipus 4 correspon a la feina d'estudiant, on s'ha considerat un salari horari corresponent al del conveni de pràctiques com a becari de l'IRI per a la feina de l'estudi previ.

Tenint en compte que aproximadament s'han dedicat de mitjana unes 25h/setmanals al projecte, s'ha fet una estimació de les hores dedicades a cada tipus de feina segons la planificació temporal del projecte, i a la Taula 6.3 es pot visualitzar el pressupost del projecte amb els costs desglossats i la contribució de cada tipus de feina al cost final.

Tipus	Concepte	Dedicació (h)	Cost horari (€/h)	Cost total (€)
Tipus 1	Disseny	100	30	3000
Tipus 2	Programació	150	25	3750
Tipus 3	Administratiu	75	15	1125
Tipus 4	Estudiant	125	8	1000
Total		450		8875

Taula 6.3: Taula de costs dels recursos humans.

Costs de recursos materials

Pel que fa als recursos materials, es poden distingir 3 tipus de despeses:

- **Recursos de hardware**: S'ha fet servir un ordinador i 2 càmeres. En ser material informàtic en un centre de recerca, s'ha tingut en compte una vida útil de 4 anys i un mètode d'amortització lineal. Es pot veure el càlcul a la Taula 6.4

Concepte	Cost unitari (€)	Vida útil	Amortització (€)
Ordinador	650	4	162,5
Webcam	24	4	6
Càmera industrial	550	4	137,5
Total			306

Taula 6.4: Taula d'amortització dels recursos de hardware.

- **Recursos de software**: Han tingut un cost de 0 € ja que s'ha fet servir programari gratuït per desenvolupar el projecte.

- **Costs indirectes**: Relacionats amb els costs de subministrament (energia elèctrica), d'instal·lacions i els costs administratius (material d'oficina). Per fer el càlcul d'aquestes despeses s'ha tingut en compte que representen aproximadament un 15% del cost total del projecte.

Costs totals

Tenint en compte el cost desglossat en els diferents tipus de despesa, s'obté el cost total del projecte que s'observa a la Taula 6.5.

S'han sumat els costs de recursos humans i de l'amortització de hardware dels recursos materials per obtenir un subtotal. A aquest valor se l'hi ha aplicat un 15% de despeses de costs indirectes i s'ha arrodonit el resultat en ser un valor aproximat.

Tipus de despesa	Cost (€)
Recursos humans	8.875
Amortització hardware	306
Subtotal	9.181
Costs indirectes (15%)	1.377
Total	10.558

Taula 6.5: Taula de cost total del projecte.

Tal i com es pot observar, el cost total del projecte és de 10.558 €.

Impacte Ambiental

En el desenvolupament d'aquest projecte, el consum de recursos materials i energètics que tenen un impacte directe sobre el medi ambient no ha estat un factor significatiu, ja que el recurs bàsic que es pot imputar de forma directe al projecte ha estat l'energia elèctrica que consumeix un ordinador, però tota acció té la seva repercussió sobre el medi per poc que sigui.

L'energia elèctrica necessària com a font d'alimentació de l'ordinador és un recurs que genera un impacte ambiental que pot ser quantificat tenint en compte que al projecte s'hi han dedicat 450h de treball. Considerant que per al consum de l'ordinador i dels monitors es necessita una potència d'uns 150W en funcionament i que en totes les hores de treball s'ha utilitzat l'ordinador i per tant consumit aquesta potència, s'obté un consum de 67,5KWh d'energia elèctrica. Tenint en compte un valor de mix elèctric peninsular de 321 g CO_2/KWh, la petjada de carboni associada al projecte en CO_2 equivalent al consum elèctric es tradueix a aproximadament 21,7 kg CO_2 [26].

Pel que fa a la resta de components utilitzats en el projecte, el material físic utilitzat era ja material existent i no es preveu la construcció de cap tipus d'artefacte, pel que no hi han intervingut processos de fabricació. El que s'ha de considerar és que en un futur s'haurà de fer un reciclatge d'aquests components que tenen una vida útil i es van deteriorant amb el temps, pel que hi haurà un impacte ambiental relacionat amb el procés de reciclatge.

Conclusions

En aquest projecte s'ha presentat la implementació d'un sistema d'SLAM Visual amb una càmera monocular basat en la resolució del problema de bundle adjustment, desenvolupant cada part involucrada en el sistema de forma separada.

En primer lloc, referent a la part més lligada a les tècniques de visió per computador, s'ha pogut comprovar que el fet de seleccionar una tríada detector-descriptor-matcher depèn completament de l'aplicació per la qual es vulgui fer servir, sent difícil escollir una opció versàtil i genèrica que sigui aplicable a casos diferents sense perdre qualitat en els resultats.

En segon lloc, pel sistema desenvolupat en aquest projecte, s'ha comprovat com pel que fa al detector i descriptor, ORB resulta l'opció més factible si es vol treballar en unes condicions similars a les presentades: fer detecció-descripció-match a temps real i realitzar el procés de forma incremental entre imatges consecutives de manera que la variació entre imatges és suficientment petita com per assumir que no fa falta considerar ni invariància a l'escala ni a la rotació. Referent al matcher, FLANNBASED és la millor elecció per tal de reduir el temps de match.

En tercer lloc, referent a la part relacionada amb Wolf, s'ha dissenyat un algoritme per fer SLAM que es basa en l'estructura de classes bases de l'arbre de Wolf, de manera que s'ha vist de manera aplicada que és una llibreria útil per resoldre problemes de localització ja que permet adaptar una estructura genèrica a les necessitats específiques de cada aplicació.

En quart lloc, s'ha presentat com resoldre el problema d'SLAM amb una càmera monocular és equivalent a resoldre el problema de bundle adjustment i s'ha implementat en C++ el codi a Wolf que permet fer aquesta tasca. S'ha implementat un algoritme que és capaç de processar les imatges que rep d'una càmera i establir tracks en imatges consecutives de forma incremental, associar cada track a un landmark de l'exterior i estimar la localització tant de la càmera com del landmark resolent un problema de graph SLAM on els factors implementen el residu característic del problema de bundle adjustment a optimitzar.

Per últim, pel que fa a l'aplicació de l'algoritme en un entorn real a través d'un node de

ROS, s'ha pogut veure com el solver convergeix a una solució a mesura que es va processant la informació rebuda però que en la visualització 3D el resultat no es correspon a la realitat ja que no es redueix prou l'error en iterar sobre l'estimació dels estats. Per una banda, la configuració del problema es crea correctament amb tots els seus elements i relacions. Per l'altra, el muntatge sobre un entorn real no proporciona encara resultats prou fiables sobre les estimacions tal i com està configurat l'algoritme. De fet és un resultat lògic perquè s'està treballant en afinar els diferents elements perquè funcioni de forma precisa.

Treball futur

Pel que fa a treball a fer a partir d'aquest projecte, per una banda es troba la millora dels resultats que s'obtenen en llançar l'aplicació sobre un node de ROS. Cal fer una revisió crítica de la configuració del problema i dels elements involucrats, ja que de forma específica i per separat desenvolupen la tasca que els correspon però integrats en un entorn real cal ajustar els resultats que presenten. Una de les tasques a fer és donar un *prior* de la pose a cada keyframe que es crea basat en una configuració semblant a la real per tal que el solver hagi de rectificar poc l'estimació i s'acosti més al moviment real. Una altra feina és millorar el calibratge de la càmera ja que en cas contrari hi pot haver errors geomètrics i no donar resultats precisos.

Per altra banda, en l'abast d'aquest projecte no s'ha integrat el sistema sobre un robot, sinó que s'ha treballat a partir del moviment d'una càmera de forma manual. Com a treball futur, hi hauria la idea d'ajuntar la feina feta en aquest sistema d'SLAM Visual amb la feina feta a l'IRI sobre una IMU i la feina feta sobre Loop Closure. D'aquesta manera, es pot explotar la llibreria Wolf, que està pensada per integrar múltiples sensors de tipologia diversa, i integrar totes aquestes feines sobre un robot per poder fer SLAM tenint els sistemes de mesura complementaris de la visió exterior i el moviment propi.

Bibliografia

[1] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment — a modern synthesis. In Bill Triggs, Andrew Zisserman, and Richard Szeliski, editors, *Vision Algorithms: Theory and Practice*, pages 298–372, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[2] J.Solà. *Simultaneous localization and mapping with the extended Kalman filter [en línia].* Octubre 2014. [Consulta: 10 de juny de 2019]. Disponible a: `http://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAMcourse.pdf`.

[3] J.Solà. *Course on SLAM [en línia].* Abril 2017. [Consulta: 10 de juny de 2019]. Disponible a: `https://www.iri.upc.edu/people/jsola/JoanSola/objectes/toolbox/courseSLAM.pdf`.

[4] J.Solà. *Towards Visual Localization, Mappingand Moving Objects Trackingby a Mobile Robot:a Geometric and Probabilistic Approach [en línia].* Febrer 2007. Tesi doctoral. Institut National Politechnique de Toulouse, Laboratoire d'Analyse et d'Architecture de Systèmes du CNRS . [Consulta: 10 de juny de 2019]. Disponible a: `www.iri.upc.edu/people/jsola/JoanSola/objectes/PhD/Thesis.pdf`.

[5] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2:31–43, 2010.

[6] Joan Solà, Jeremie Deray, and Dinesh Atchuthan. A micro Lie theory for state estimation in robotics. *arXiv e-prints*, page arXiv:1812.01537, Dec 2018.

[7] A.Kaehler i G.Bradski. *Learning OpenCV 3 : Computer Vision in C++ with the OpenCV Library.* O'Reilly Media, 1st edition, Desembre 2016. Capítol 16. ISBN 9781491937990.

[8] Elmar Mair, Gregory Hager, Darius Burschka, Michael Suppa, and Gerhard Hirzinger. Adaptive and generic corner detection based on the accelerated segment test. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial*

Intelligence and Lecture Notes in Bioinformatics), volume 6312 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 183–196, 2010.

[9] David G Lowe. Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60(2):91–110, 2004.

[10] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 430–443, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. Computer Vision–ECCV 2006, pages 404–417, 2006.

[12] Shi, J. and Tomasi, C.1994. Good features to track. InProc. IEEE Int. Conf. on Computer Vision andPattern Recognition. Seattle, USA, 593–600.

[13] Pablo Fernández Alcantarilla, Adrien Bartoli, and Andrew J Davison. Kaze features. In Computer Vision–ECCV 2012, pages 214–227. Springer, 2012.

[14] Harris, C. and Stephens, M.1988. A combined corner and edge detector. InFourth Alvey VisionConference. Manchester (UK).

[15] Pablo F Alcantarilla, Jesús Nuevo, and Adrien Bartoli. Fast explicit diffusion for accelerated features in nonlinear scale spaces. Trans. Pattern Anal. Machine Intell, 34(7):1281–1298, 2011.

[16] Per-Erik Forssén. Maximally stable colour regions for recognition and matching. In Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on, pages 1–8. IEEE, 2007.

[17] Stefan Leutenegger, Margarita Chli, and Roland Yves Siegwart. Brisk: Binary robust invariant scalable keypoints. In Computer Vision (ICCV), 2011 IEEE International Conference on, pages 2548–2555. IEEE, 2011.

[18] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua, "BRIEF: Binary Robust Independent Elementary Features", 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.

[20] E. Tola, V. Lepetit, and P. Fua. DAISY: An Efficient Dense Descriptor Applied to Wide Baseline Stereo. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(5):815–830, May 2010.

[21] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst. Freak: Fast retina keypoint. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, pages 510–517. Ieee, 2012.

[22] Gil Levi and Tal Hassner. LATCH: Learned Arrangements of Three Patch Codes. IEEE Winter Conference on Applications of Computer Vision (WACV), Lake Placid, NY, USA, 2016.

[23] Eric Christiansen David Kriegman Ziegler, Andrew and Serge J. Belongie. Locally uniform comparison image descriptor.

[24] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[25] Joan Solà, Teresa Vidal-Calleja, Javier Civera, Jose Maria Martinez-Mont. Impact of landmark parametrization on monocular EKF-SLAM with points and lines. International Journal of Computer Vision, Springer Verlag, 2011, 97 (3), pp.339-368. 10.1007/s11263-011-0492-5. hal-00451778v4. [Consulta: 11 de juny de 2019]. Disponible a: `https://hal.archives-ouvertes.fr/hal-00451778/file/solaIJCV11.pdf`.

[26] Oficina Catalana del Canvi Climàtic. Nota informativa sobre la metodologia de estimación del mix eléctrico por parte de la oficina catalana del cambio climático [en línia], Febrer 2019. [Consulta: 9 de juny de 2019]. Disponible a: `http://canviclimatic.gencat.cat/web/.content/04_ACTUA/Com_calcular_emissions_GEH/factors_emissio_associats_energia/190219_Nota-metodologica-mix_esp.pdf`.